Abstract Execution: Automatically Proving Infinitely Many Programs

Abstrakte Ausführung: Automatisches Beweisen unendlich vieler Programme
Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
vorgelegte Dissertation von Dominic Steinhöfel aus Frankenthal/Pfalz
Tag der Einreichung: 27. Februar 2020, Tag der Prüfung: 8. Mai 2020

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Prof. Gilles Barthe, Ph.D.
Darmstadt – D 17

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science
Department
Software Engineering Group

Abstract Execution: Automatically Proving Infinitely Many Programs
Abstrakte Ausführung: Automatisches Beweisen unendlich vieler Programme

Doctoral thesis by Dominic Steinhöfel

1. Review: Prof. Dr. Reiner Hähnle
2. Review: Prof. Gilles Barthe, Ph.D.

Date of submission: 27. Februar 2020
Date of thesis defense: 8. Mai 2020

Darmstadt – D 17

Erklärungen laut Promotionsordnung

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 27. Februar 2020

D. Steinhöfel

Acknowledgments

It is good tradition to start an acknowledgments section of a thesis written in Reiner Hähnle's Software Engineering Group by a description of Reiner's contributions not only to the thesis as a supervisor, but also to its author as a friend. Everything that has been said is true and applies to me and my thesis, too. Reiner is an amazing supervisor, who was always available when needed. By never losing the overview, he helped me to steer my work into the right directions. He is also a great coauthor. Not only is it a pleasant experience to write on a paper together with him; It also improved my style in many ways. I really enjoyed the occasions when Reiner—always being very generous—invited the group to his home or Vino Central, and the conference stays, where occasionally, the beers have double the expected size. Speaking of Vino Central, Reiner contributed significantly to my evolving taste for excellent wines (and, to a more limited degree, for *really* expensive fish). Finally, I cannot but mention Reiner's outstanding leadership qualities. There are many group leaders in the scientific community thinking that only fine-grained micro management enables high-quality research. Reiner constantly proves them wrong. He established a research environment based on mutual support, friendship and solidarity. It works: Group members publish, advance the state of research, even graduate in time; and all this while feeling well and respected. Thanks, Reiner, and stay as you are.

I would like to express my gratitude to Gilles Barthe for having agreed to be my second reviewer and for his valuable feedback and suggestions for further applications of AE during my defense.

A number of people voluntarily agreed to give me, on shortest notice, feedback to selected parts of this thesis. In the order of the chapters they reviewed: Nathan Wasser (The Loop Scope Method and Completion Scopes), Richard Bubel (Abstract Execution), Hans-Dieter Hiep and Michaël Lienhardt (Modal and Symbolic Trace Logic), and Volker Stolz (Correctness of Refactoring Techniques). Thank you all very much for your help, and for discovering both stupid and subtle problems in the writing. Specifically, I want to thank Nathan for always discovering the most subtle problems related to the semantics of Java, especially in symbolic execution rules and program transformation, Richard for giving last-minute feedback during the weekend, Hans-Dieter for the most interesting feedback style I've ever seen, and Michaël for being amazingly responsive and looking up even the definition of single words of the prose. In addition, I would like to thank Volker for giving me the opportunity to present REFINITY during the refactoring tutorials at iFM'19.

Working in the Software Engineering Group has always been great, thanks to the great people who are or used to be working there. I would like to name some of them specifically: Nathan Wasser, one of the supervisors of my Master's thesis, who made me one of the godfathers of his cute daughter, and with whom I've been having the funniest email conversations. Martin Hentschel, who showed me that also people from northern Germany can be friendly and humorous, and who's a secret party king. Antonio Flores Montoya, with whom you can have a *lot* of fun, and who joined us for epic (trash) parties. Huy Quoc Do, for being a *little* bit disappointed, and for being a bag full of miracles. Asma (without trailing "e") Heydari Tabar for being funny (Laternchen) and always helpful. Eduard Kamburjan, for the intense discussions (eventually you will learn what a tea is), the nice cultural evenings, the famous classic vs. trash movie evenings, the ice cream, the cappuccinos, and the endless jokes. Stefan Dillmann for tolerating all this. Last but not least, I would like to thank Priv.-Doz. Dr. Richard Bubel, who also supervised my Master's thesis, for being one of the most altruistic, helpful and friendly persons I ever met, and the only other social democrat. We all owe you a lot.

Diese Dissertation wäre nicht möglich gewesen ohne meine wunderbare Frau Julia. Sie baute mich stets auf zwischen den manischen Phasen während des Schreibens, ertrug Panikattacken, schlechte Laune und Zweifel und half stoisch-positiv bei deren Überwindung. Dabei gelang es ihr fast immer, nicht zu erwähnen, dass es doch gut wäre, wenn all dies endlich vorbei sei. Während ich teilweise 14 Stunden täglich an diesem Dokument schrieb, kümmerte Julia sich um unsere wundervolle Tochter Mia, das süßeste, intelligenteste, hübscheste, und lauteste Baby der Welt. Ich liebe euch über alles!

Abstract

Abstract programs contain schematic placeholders representing potentially infinitely many concrete programs. They naturally occur in multiple areas of computer science concerned with correctness: rule-based compilation and optimization, code refactoring and other source-to-source transformations, program synthesis, Correctness-by-Construction, and more. Mechanized correctness arguments about abstract programs are frequently conducted in interactive environments. While this permits expressing arbitrary properties quantifying over programs, substantial effort has to be invested to prove them *manually* by writing proof scripts. Existing approaches to proving abstract program properties *automatically*, on the other hand, lack expressiveness. Frequently, they only support placeholders representing *all possible instantiations*; in some cases, minor refinements are supported.

This thesis bridges that gap by presenting *Abstract Execution*, an *automatic* reasoning technique for universal behavioral properties of abstract programs. The restriction to universal (no existential quantification) and behavioral (not addressing internal structure) properties excludes certain applications; however, it is the key to automation. Our logic for Abstract Execution uses abstract state changes to represent unknown effects on local variables and the heap, and models abrupt completion by symbolic branching. In this logic, schematic placeholders have *names*: It is possible to re-use them at several places, representing the same program elements in potentially different contexts. Furthermore, the represented concrete programs can be constrained by an expressive *specification language*, which is a unique feature of Abstract Execution. We use the theory of *dynamic frames* to scale between full abstraction and total precision of frame specifications, and support fine-grained pre- and postconditions for (abrupt) completion.

We implemented Abstract Execution by extending the program verifier KeY. Specifically for relational verification of abstract Java programs, we developed REFINITY, a graphical KeY frontend. We used REFINITY it in our signature application of Abstract Execution: to model well-known statement-level *refactoring techniques* and prove their conditional safety. Several yet undocumented *behavioral preconditions for safe refactorings* originated in this

case study, which is one of very few attempts to statically prove behavioral correctness of *statement-level* refactorings, and the only one to cover them to that extent.

Abstract Execution extends Symbolic Execution for abstract programs. As a foundational contribution, we propose a *general framework for Symbolic Execution* based on the semantics of symbolic states. It natively integrates state merging by supporting m-to-n transitions. We define two orthogonal correctness notions, *exhaustiveness* and *precision*, and formally prove their relation to program proving and bug detection.

Finally, we introduce Modal Trace Logic (MTL), a trace-based logic to represent a variety of different program verification tasks, especially for relational verification. It is a "plug-in" logic which can be integrated on-demand with formal languages that have a *trace semantics*. The core of MTL is the *trace modality*, which allows expressing that a specification *approximates* an implementation after a *trace abstraction* step. We demonstrate the versatility of this approach by formalizing concrete verification tasks in MTL, ranging from functional verification over program synthesis to program evolution. To reason about MTL problems, we translate them to *symbolic traces*. We suggest Symbolic Trace Logic (STL), which comes with a sequent calculus to prove symbolic trace inclusions. This requires checking symbolic states for subsumption; to that end, we provide two generally useful notions of *symbolic state subsumption*. This framework relates as follows to the other parts of this thesis: We use the language of *abstract programs* to express synthesis and compilation, which connects MTL to Abstract Execution. Moreover, symbolic states of STL are based on our framework for Symbolic Execution.

Zusammenfassung

Abstrakte Programme beinhalten Platzhalter, welche unendlich viele konkrete Programme repräsentieren können. Sie werden in verschiedenen, sich mit der Korrektheit von Programmen befassenden Bereichen der Informatik verwendet: Regelbasierte Kompilierung und Optimierung, Refaktorisierung von Programmen und andere Quelltexttransformationen, Programmsynthese, inkrementell-korrektheitserhaltende Konstruktion von Programmen[1], und mehr. Mechanisierte Korrektheitsargumente für abstrakte Programme werden häufig in interaktiven Beweisumgebungen durchgeführt. Dies erlaubt den Ausdruck beliebiger, über Programme quantifizierender Eigenschaften; allerdings ist es sehr aufwändig, diese *manuell* durch das Schreiben von Beweisskripten zu verifizieren. Bestehende *automatische* Beweissysteme für abstrakte Programme leiden hingegen unter einer geringen Ausdrucksstärke. Häufig unterstützen diese Systeme ausschließlich Platzhalter, welche *alle möglichen Programme* repräsentieren; in einigen Fällen werden geringe Verfeinerungen unterstützt.

Die vorliegende Dissertation schließt diese Lücke durch das Konzept der *abstrakten Ausführung*, einer *automatischen* Beweistechnik für universelle Verhaltenseigenschaften abstrakter Programme. Die Beschränkung auf universelle (keine existentiellen Quantoren) und *Verhaltens*eigenschaften (welche sich nicht auf die innere, syntaktische Struktur der Platzhalter beziehen) schließt gewisse Anwendungen aus; allerdings ermöglicht dies erst den hohen Automatisierungsgrad. Die Logik abstrakter Ausführung verwendet abstrakte Zustandsübergänge, um unbekannte Effekte auf lokale Programmvariablen und den Haufenspeicher[2] zu modellieren; für abrupte Terminierung werden symbolische Fallunterscheidungen durchgeführt. Platzhalter haben *Namen*: Dies erlaubt ihre Verwendung an verschiedenen Stellen, wo sie die selben Programmelemente in möglicherweise unterschiedlichen Kontexten repräsentieren. Zusätzlich können Instantiierungen durch eine ausdrucksstarke *Spezifikationssprache* begrenzt werden. Diese ist ein auszeichnendes Merkmal abstrakter Ausführung. Der Einsatz der Theorie dynamischer Rahmen[3] ermöglicht

[1] engl. *Correctness-by-Construction*
[2] engl. *heap*
[3] engl. *dynamic frames*

es, beschreibbare Speicherbereiche beliebig präzise anzugeben. Das System unterstützt Nachbedingungen sowie feingranulare Vorbedingungen für abrupte Terminierung.

Abstrakte Ausführung ist im Programmbeweiser KeY implementiert. Speziell für die Verifikation relationaler Programmeigenschaften wurde REFINITY entwickelt, eine grafische Benutzeroberfläche als Erweiterung von KeY. Diese wurde in der bisher wichtigsten Anwendung abstrakter Ausführung eingesetzt: der Modellierung bekannter *Refaktorisierungstechniken* auf Anweisungsebene[4], deren Korrektheit unter Bedingungen bewiesen wurde. Diese Fallstudie brachte mehrere, bisher undokumentierte Vorbedingungen für die sichere[5] Anwendung von Refaktorisierung hervor, und ist damit einer der wenigen Versuche, statisch die (semantische) Korrektheit von Refaktorisierungstechniken auf Anweisungsebene zu beweisen, und der einzige, welcher es in diesem Umfang vermochte.

Als Beitrag zum formalen Fundament abstrakter Ausführung enthält diese Arbeit eine allgemeine Theorie *symbolischer Ausführung* basierend auf der Semantik symbolischer Zustände. Zustandsverschmelzung[6] wird direkt unterstützt, da das formale System auf m-zu-n-Zustandsübergängen beruht. Ein wichtiger Beitrag sind zwei orthogonale Korrektheitsbegriffe, *Abdeckung*[7] und *Präzision*, deren Zusammenhang zu deduktiven Programmbeweisen und automatisierter Fehlersuche formal bewiesen wird.

Schließlich beschreibt diese Ausarbeitung die modale Zustandssequenzlogik[8] (MTL), eine auf Zustandssequenzen[9] beruhende Logik zur Repräsentation unterschiedlicher Programmverifikationsprobleme, insbesondere für relationale Verifikation. Es handelt sich um ein flexibles System, welches nach Bedarf um formale Sprachen mit Zustandssequenzsemantik erweitert werden kann. Der Hauptbestandteil von MTL ist die *Zustandssequenzmodalität*[10], welche ausdrückt, dass eine Spezifikation eine Implementierung nach einem *Abstraktionsschritt approximiert*. Die Vielseitigkeit dieses Formalismus wird durch die Anwendung auf eine Reihe von Problemfeldern demonstriert, darunter u.A. funktionale Verifikation, Programmsynthese und Programmevolution. Um MTL-Probleme mechanisiert zu beweisen, übersetzen wir sie in symbolische Zustandssequenzen. Speziell für diese wurde die symbolische Zustandssequenzlogik[11] (STL) eingeführt, welche über einen Sequenzenkalkül zum Beweisen von Teilmengeneigenschaften symbolischer Zu-

[4] engl. *statement-level*
[5] engl. *safe*
[6] engl. *state merging*
[7] engl. *exhaustiveness*
[8] engl. *Modal Trace Logic*
[9] engl. *traces*; Aufzeichnungen während einer Programmausführung anfallender Zustände. Wird auch mit *Spuren* übersetzt.
[10] engl. *trace modality*
[11] engl. *Symbolic Trace Logic*

standssequenzen verfügt. Dieser erfordert die Analyse von symbolischen Zuständen auf „Subsumption“. Die Basis dafür besteht in zwei allgemeinen, neuen Subsumptionsbegriffen. MTL und STL verhalten sich wie folgt zu den restlichen Bestandteilen dieser Dissertation: Um Programmsynthese und Kompilierung zu formalisieren, wird die Sprache *abstrakter Programme* verwendet, was MTL mit abstrakter Ausführung verbindet. Weiterhin basieren symbolische Zustände in STL auf der Theorie symbolischer Ausführung.

Contents

List of Figures

List of Tables

List of Listings

List of Definitions

List of Lemmas, Propositions, Theorems

List of Acronyms

My desire and wish is that the things I start with should be so obvious that you wonder why I spend my time stating them. This is what I aim at because the point of philosophy is to start with something so simple as not to seem worth stating, and to end with something so paradoxical that no one will believe it.

Bertrand Russell
The Philosophy of Logical Atomism

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.

Bertrand Russell
The Conquest of Happiness

1 Introduction

> However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs.
>
> *C.A.R. Hoare* [Hoa69]

Deductive program verification [Fil11; HH19] aims to prove that a program accomplishes its user's intentions by purely deductive reasoning, i.e., the application of valid rules of inference. Since the early work on deductive program verification in the late 1960s [Flo67; Hoa69], there have been numerous advances in this area. The studied verification tasks no longer merely focus on proving partial functional correctness. Basic variations include termination [Häh+86], reachability [RHS95], and program synthesis [Hei92; Smi90; SGF10]. Starting in the early 2000s, the verification of *relational* properties relating different programs, different versions of the same program, or the same program for different inputs [BU18] has come into focus. Example properties are information flow [DHS05; SM03], correct compilation [Kum+14; Ler09], correctness of program transformations (refactoring) [GM06], or program evolution [GS13]. Different verification techniques have been developed to address these properties, comprising dynamic logic [HTK00], relational Hoare logic [Ben04], Hoare quadruples [Yan07], self composition [BDR04; DHS05], product programs [BCK11], and more.

Formal verification of nontrivial, let alone "realistic" programs, is tedious and error-prone [Ahr+16]. Luckily, formal deductive proofs of the correctness of a program written in Java or C do not have to be conducted by hand these days. Not only the regarded research problems and developed scientific theories evolved, but also the tool support for applying the aforementioned methodologies. Frequently, such tools are based on general-purpose interactive theorem provers, currently mainly Coq [Dow+93] or Isabelle [NPW02]. Examples include the Verified Software Toolchain for C written in Coq [App12] or a formalization of sequential Java in Isabelle [Str02]. Apart from that, many dedicated program provers have been developed: VCC [Dah+09], Dafny [Lei10], Frama-C [Cuo+12], VeriFast [VJP15], OpenJML [Cok14] and KeY [Ahr+16], to name a few examples. These tools are capable of proving complex properties about programs in industrial programming languages such as C, C# and Java, as demonstrated in, e.g., [Ler09; Dah+09; PL10; Kum+14; Gou+19]. Thus, one may argue that the second part of above quotation by Tony Hoare no longer holds true: Program proving, as of now, *is* applicable to non-simple designs. Considered program designs comprise elaborate setups like compilers [Ler09; Kum+14] or the complicated sorting routine for nontrivial types in the Java standard library [Gou+19]. Notwithstanding, the first part of the quotation still holds. The situation that *programmers* conduct tool-supported *formal proofs* of their systems is not yet in reach; formal proofs are still mostly conducted by highly skilled experts. Verifying two complex subroutines in [Gou+19], carried out by such experts, nevertheless took about *three person-months*. This number is all the more impressive if one takes into account that in the mentioned case study, merely the absence of unexpected runtime exceptions was shown, and not that the algorithm is actually functionally correct. As one of the main bottlenecks of, in particular, functional verification, the lack of *specifications* has been identified [Bau+12; BB13; Gra15; Gou+19; Ahr+16]. It is basically impossible to prove the correctness of arbitrary Java programs without a full functional specification of the Java standard library. Furthermore, the necessity of loop invariants and method contracts to reason symbolically about looping and recursive programs constitutes a major problem, which impedes that program reasoning techniques are taken up by “mainstream” programmers.

Is formal program verification therefore confined to toy programs, or if anything to isolated problems addressed by a team of specialists dedicating months of work to the proof of some routines? Luckily, this is not the case; at least not if we are willing to go beyond full *functional* verification of *individual* programs. For instance, in relational verification, it is not necessary to provide complete functional specifications of the target programs. Instead, their respective behaviors are compared; loops are linked by *coupling invariants*, which are structurally simpler than functional loop invariants and can frequently be inferred automatically [BU18]. Another well-known application are *verified compilers* [Ler09; Kum+14]. The implementation, specification and proof of these compilers is performed by a team of experts, but after that, the extracted compiler is an executable that can be used even by average programmers. By compiling programs with a verified compiler, one obtains a guaranty that the compiled program is functionally equivalent to its source. When instead choosing the way of functional verification, the programmer has to come up with a full specification of the program, including loop invariants, contracts for library methods, etc., and to prove the source program *and* the target program correct. Most

Listing 1.1: Before

```
if (\abstract_expression e) {
  \abstract_statement Q1;
  \abstract_statement P;
} else {
  \abstract_statement Q2;
  \abstract_statement P;
}
```

Listing 1.2: After

```
if (\abstract_expression e) {
  \abstract_statement Q1;
} else {
  \abstract_statement Q2;
}

\abstract_statement P;
```

Figure 1.1: Example Application of Abstract Execution—Moving a Common Postfix to After a Conditional Statement

likely, even different provers will have to be used, since source and compiled program are expressed in different languages. This has to be done *for every compiled program*. The key difference is consequently that in functional verification, one obtains a result for all possible input *values*, whereas in case of a verified compiler, the desired property additionally ranges over input *programs*. In other words, instead of proving a first-order property, one proves a *second-order property*, or alternatively, a *"once-and-for-all" result*.

The best-known way to prove such properties, apart from pen-and-paper proofs, is to use a general-purpose interactive proof assistant. This thesis proposes *Abstract Execution (AE)* [SH19a] which allows to *automatically* prove second-order properties about sequential programs of an industrial programming language with side effects and abrupt completion.

For example: If all legs of a conditional statement have a *common* postfix, the postfix may be moved to after the conditional. AE allows formalizing source and target of the described transformation as *abstract programs*, i.e., programs with *schematic* statements and expressions. Equivalence of the abstract program model can be proven for all *"legal instantiations"* of schematic placeholders. We call these placeholders *Abstract Program Elements (APEs)*. The concrete syntax to declare them is "`\abstract_statement P;`" and "`\abstract_expression e;`", where `P`, `e` are identifiers for the abstract statement and expression. Intuitively, two APEs with the same identifier symbol represent the same concrete instantiations. Figure 1.1 shows the abstract programs for this example before and after the transformation.

The term Abstract Execution is short for "symbolic *execution* of *abstract* programs". The key to automation is to use *abstract state transitions* abstracting away from the effects of an APE on the symbolic execution state. Automation comes at the price of expressivity: We can only express a limited degree of second-order inference. This excludes, for instance, second-order induction and higher-order quantification, but also existential properties and properties constraining the *syntactic structure* of instantiations for APEs. We address *universal* and *behavioral* properties. In particular, we are concerned with the *external effects* of APEs on the program state, and are aware of abrupt completion.

Considering expressivity, we outperform prior approaches for automatic verification of abstract programs (e.g., [ARS05; BRR08; KTL09; GS13; Lop+18; SH18]). APEs can be augmented with a *fine-grained specification* of their frame (locations they may write to), footprint (locations they may depend on), and preconditions for abrupt completion. Additionally, it is possible to provide a functional specification of the resulting state. We trade off abstraction and precision using the theory of *dynamic frames* [Kas11], which introduces specification variables for sets of locations. For example, the frame of an APE can be given an abstract name, which can be reused for other APEs. Moreover, we support the specification of constraints expressing, e.g., disjointness of different dynamic frame specification variables. Behaviors of different APEs can be *coupled*. Thus, one can express, e.g., that an abstract statement throws an exception if, and only if, some abstract expression evaluates to a certain value.

Refactoring is the process of changing code in a way that does *not alter its external behavior*, yet improves its *internal structure* [Fow18]. We apply AE to prove behavioral equivalence of nine statement-based refactoring techniques, including two with loops. Our general assumption is that input and output programs compile, and that there are no issues with name binding and accessibility. For almost all refactorings, we found non-trivial preconditions which have to be satisfied for a safe application of the techniques. Most of them have not yet been mentioned in literature.

In deductive program verification, problems to be solved are usually expressed in natural language and then *directly* formalized in the solution approach. This leads to the already mentioned diversity of verification formalisms and techniques developed for specialized research areas. We regard this as problematic for two reasons. First, there is a natural tendency to use a specific solution approach just because it is familiar, and not because it is best suitable. And second, it is hard to detect similarities and to transfer insights from one problem area to another if solution methods are already formalized in specific frameworks of potentially very different flavor. For example, even though the fields of model checking and deductive program verification are converging [Sha18], the practical interaction between the communities is still sparse. The same holds for the interaction between the abstract interpretation and deductive program verification communities. We propose Modal Trace Logic (MTL), an *abstract semantic framework* to express a wide range of program verification problems. The only mandatory syntactic element of MTL is the *Trace*

Modality, allowing to relate objects of different languages, like programs, postconditions or temporal logic formulas, in a uniform way. MTL can be syntactically enriched by "plugging in" additional Trace Description Languages (TDLs). A TDL is any formal language with an explicit trace semantics. The trace modality is particularly well-suited for applications in relational program verification, since we can use a program as a *specification* of another program. To demonstrate the versatility of MTL, we formalize various program verification problems in the logic. Some formalizations, e.g., program synthesis and compilation, require *abstract programs*, thus relating MTL to Abstract Execution.

To reason about problems specified in MTL, we propose Symbolic Trace Logic (STL), a logic based on *regular symbolic traces*. The idea behind STL is that programs as well as formulas are translated to symbolic traces. Atoms of our regular symbolic trace language are *symbolic states*; an important ingredient of STL is the definition of the semantics of symbolic states as their *concretizations*, i.e., the concrete states represented by them. Based on this foundational concept, we define a *universal theory* of Symbolic Execution (SE). Our theory is more general than other formal definitions of SE [Kne91; LRA17; BB19]. Instead of defining correctness of SE building on simulation relations between symbolic and concrete transition systems, our notions of *exhaustiveness* and *precision* of symbolic transitions are solely based on the effects of the transitions on the semantics of input and output states. The framework has general m-to-n transitions, and therefore readily supports symbolic state merging.

1.1 State of the Art

Approaches *automatically* proving problems involving universal quantification over programs usually prove the correctness of transformation *rules* (e.g., in the area of (optimizing) compilation or symbolic execution) [ARS05; BRR08; KTL09; Lop+18]. The most prominent work in an explicitly relational context is on *automated regression verification* [GS13]. These techniques have the following common characteristics:

- They aim to prove the correctness of *concrete* rules or programs, which are treated using post-hoc abstraction (e.g., to handle schematic parts of rules or recursive method calls). This is contrary to our goal to support *abstract modeling*. Therefore, these approaches do neither support nor require additional fine-grained specifications.
- Abrupt completion and/or side effects of schematic elements are frequently disregarded (e.g., [GS13] does not model either).
- With the exception of [KTL09], only abstract statements *or* abstract expressions are

supported. Reference [KTL09], on the other hand, requires lockstep execution.

The state of the art in abstract program proving focuses on proving problems of inherently small size (schematic rules) or with a restricted notion of schematic entity (regression verification). Tools used comprise introduction of uninterpreted functions [GS13] or Skolem variables with unknown side effects [BRR08], bisimulation and correlation relations [KTL09], projection to a term-rewriting system using an existing embedded theory [ARS05], and restriction to a small Domain-Specific Language which can be projected to an input for an an SMT solver [Lop+18].

The correctness of refactorings has been studied in literature; however, proven-correct static results are sparse. The most comprehensive endeavors providing such results focus on *non-behavioral* properties [Sch+12; SSM15], i.e., naming and accessibility. These approaches introduce an additional layer (a Java dialect or Alloy models) where the studied problems cannot appear due to restrictions enforceable by the type checker. One work proves correctness of three refactorings above statement level, e.g., *Pull Up/Push Down Method* [GM06], by modeling them in a term rewriting system. Others target runtime enforcement and do not elicit new refactoring preconditions (e.g., [Soa+10; EBS16]).

There is less work on formalizations of Symbolic Execution, and even less on frameworks unifying program verification problems. Considering the former, SE is defined based on simulation relations between concrete and symbolic transition systems. [Kne91; LRA17; BB19] All of these approaches provide comparable "correctness" notions (e.g., *precision* and *coverage*). Systems [LRA17; BB19] do not discuss the semantics of symbolic states; the definition in [Kne91] is not very intuitive. Regarding the latter research area, we do not know of theoretical frameworks specifically targeting the unification of program verification problems. The concept of *asynchronous product programs* [BCK13], however, comes close: It permits relating two (potentially nondeterministic) programs, which may be of *different languages* and have *different termination behavior*. Correctness of two programs w.r.t. a relational specification is reduced to *functional* correctness of a "left product" of the programs. Nondeterminism is modeled by abstract functions as in [GS13]. While asynchronous product programs allow for a certain unification of verification problems, this is not their main intention, which is a practical one: to project relational to functional program correctness. The approach *exclusively* addresses relational problems, and does not incorporate an abstraction step, e.g., as in abstract interpretation.

1.2 Contributions

This dissertation defines a Java dialect for *abstract* programs, including a specification language extending the Java Modeling Language. We define syntax and semantics of abstract programs, and provide SE rules for abstract statements and expressions. We furthermore extend the theory of dynamic frames, introduce a notion of *abstract updates* modeling abstract state changes, and define semantics and provide calculus rules for both. The framework is *implemented* for the SE engine of the KeY system. We extended KeY's DSL for calculus rules (taclet language) to support the definition of AE rules as taclets; furthermore, we added the first taclet-based loop invariant rule, which is also AE-aware. Before, loop invariant rules were implemented as "built-in rules" directly in Java.

To support the construction of "abstract program models", in particular for relational verification, we created REFINITY, a graphical KeY frontend with special support for abstract programs. We evaluate REFINITY by defining abstract program models for statement-based refactorings. For almost all refactorings, we discovered non-trivial behavioral preconditions (via failed proof attempts) which have to be satisfied to safely apply a refactoring. Occasionally, there are several variants of a refactoring technique, or several combinations of preconditions guaranteeing safety, which we also discuss. To prove refactorings with loops, we introduce the notion of *abstract strongest loop invariants*. We address abruptly completing loop bodies by an extension of the latter with the catchy name *strongest* abstract strongest loop invariants. These invariants also have to be satisfied by abstract loop bodies in case of abrupt completion.

Contributing to a unification of different program verification problems and techniques, we define Modal Trace Logic, an extensible logic with a trace semantics. We formalize several verification problems in this framework, from functional verification over program synthesis to program evolution. To facilitate reasoning about MTL formalizations, we propose to translate MTL expressions to regular symbolic traces. Our logic dedicated to symbolic traces, Symbolic Trace Logic, has a sequent calculus to reason about inclusion relations between trace sets represented by regular symbolic trace expressions.

Further contributions comprise a new theoretic framework for Symbolic Execution based on the semantics of symbolic states, a revised account of the semantics of "loop scopes" and a refreshed loop scope-based loop invariant rule, and the first description of the novel concept of "completion scopes" which are designed to control abrupt completion in SE.

1.3 Overview of Publications

The papers to which I contributed prior to finishing this thesis are listed below. They are divided into two categories: Papers whose contents are largely included in the thesis, and others. All the mentioned publications influenced this thesis to some extent. Each item contains a short description of the work, its relation to the thesis, and the extent of my contribution. Inside the categories, they appear chronologically.

Publications Included in This Thesis

- *A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows* (IFM 2017) [SW17] [Main Author]: The loop scope method, presented in Sect. 2.3, originated from the PhD thesis of Nathan Wasser [Was16]. I contributed to this paper by mostly writing it, implementing the loop scope invariant rule in KeY, and conducting a case study. Sect. 2.3 contains an improved definition of the semantics of loop scopes and updated calculus rules. Moreover, I briefly describe the new implementation of the loop scope invariant rule in KeY's *taclet* language and a repetition of the case study of [SW17] with this implementation, and contrast the results with new data collected from proofs with activated one-step simplification.
- *Abstract Execution* (FM 2019) [SH19a] [Main Author]: The original paper on AE, which introduces our methodology of specifying and proving abstract programs. The approach is applied to prove the correctness of refactoring techniques. Chapter 4 describes the principles of AE. Compared to [SH19a], I added abstract expressions, extended the specification language, based the framework on the theory of dynamic frames, added the possibility to specify functional postconditions, improved the SE rules for APEs, and completely replaced the simplification rules for abstract updates. In Appendix B, I provide a new correctness proof for AE rules. The application to refactoring of Java programs is discussed in Chapter 6. I added one more refactoring technique (*Consolidate Conditional Expression*), discuss discovered preconditions in much more detail, and replaced the approach to reasoning about abstract programs with loops. In contrast to [SH19a], the new approach is fully automatic, and does not require interactive or scripted loop coupling.
- *The Trace Modality* (DaLí 2019) [SH19b] [Main Author]: Introduces the notion of the trace modality and a reasoning system based on simulations on first-order symbolic automata. The paper uses fundamental principles of AE. In this thesis, the trace modality appears in Chapter 5 as the main constituent of a larger framework

called Modal Trace Logic (MTL). MTL is a "plugin" logic, which can be integrated with different languages as long as they have a trace semantics. Instead of the reasoning system sketched in [SH19b], I describe Symbolic Trace Logic (STL), a logic for regular symbolic traces with a sequent calculus. To integrate MTL and STL, I specify constraints on translations of MTL expressions to symbolic traces.

Other Publications

- *A General Lattice Model for Merging Symbolic Execution Branches* (ICFEM 2016) [SHB16] [Main Author]: This paper proposes a lattice-based framework for merging states in symbolic execution trees, thus mitigating the state explosion problem. For merge techniques complying with the framework, a general correctness result is obtained. Chapter 3 contains a simpler and more general theory of symbolic execution, subsuming [SHB16]. Sect. 3.3 presents concrete state merging techniques also used in [SHB16] as well as some others.
- *Modular, Correct Compilation with Automatic Soundness Proofs* (ISoLA 2018) [SH18] [Main Author]: Proposes to project the correctness of compilation rules of a rule-based compiler from LLVM IR to Java to "justifying" formulas which can be proven automatically. To account for schematic parts of compilation rules, this approach uses an early and lightweight notion of AE, inspiring the current variant. The paper uses a binary modality for relating source and compiled program subject to a set of observable variables, which inspired our trace modality.
- *Verifying OpenJDK's Sort Method for Generic Collections* (J. Automated Reasoning 62(1)) [Gou+19]: An extended journal version of paper [Gou+15] in which the authors discovered a bug in the sorting algorithm for collections of non-trivial type implemented in OpenJDK and devised a correction. In this earlier work, they could not prove the correctness of the whole corrected algorithm with KeY, since two big methods were out of reach due to the state explosion problem of symbolic execution. I am coauthor of Section 6.3 in the journal version, which, among other things, describes how the state merging framework of [SHB16] is used to prove those methods. Also, I am the main author of Section 5 providing a statistical analysis of the whole proof. The TimSort case study is the currently largest one conducted with KeY and demonstrates the efficacy of (our approach to) state merging.

1.4 Structure of This Thesis

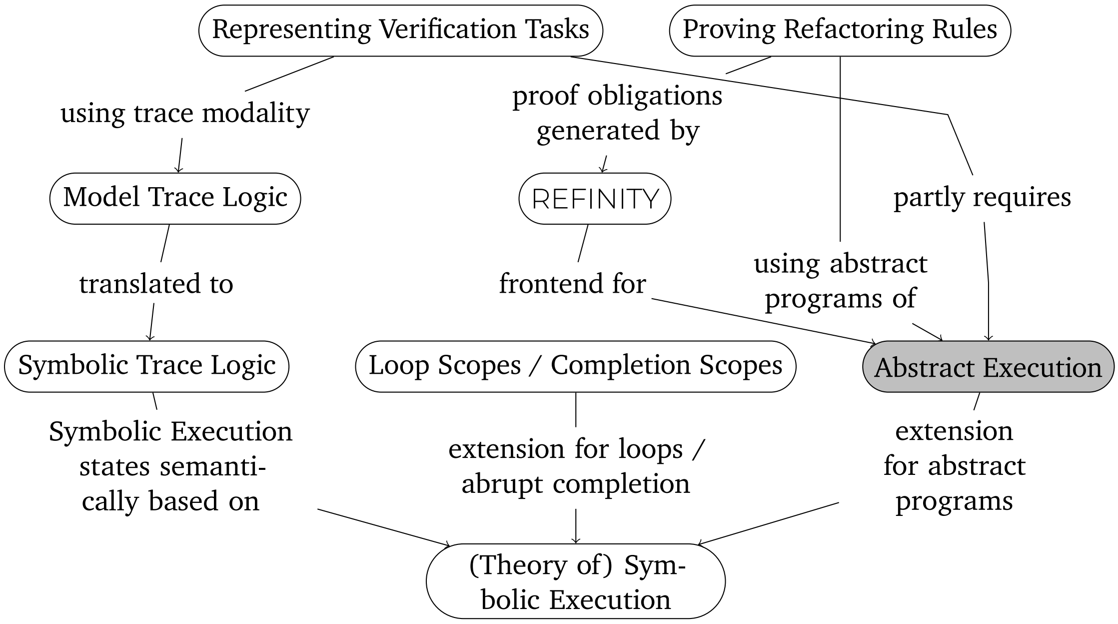


Figure 1.2: Dependencies Between Parts of This Thesis

Figure 1.2 depicts the dependencies between the parts of this thesis. The graph misses some less important edges, e.g., we need loop scopes to execute abstract programs with loops.[1] The remainder of this document is organized as follows:

Chapter 2: Theoretical Foundations Contains basic notational definitions and a primer to Java Dynamic Logic (JavaDL) and Java Modeling Language. Two sections are devoted to loop scopes and completion scopes. Finally, we sketch a variant of JavaDL with quantifiers over programs.

Chapter 3: A Theory of Symbolic Execution In this chapter, we define our theoretic framework for SE, including a section on state merging.

Chapter 4: Abstract Execution This is the main chapter of the thesis. It introduces the

[1] Prior loop invariant rules relied on intricate program transformations, leading to complications in conjunction with APEs, which are *atomic* and cannot easily be transformed. Using loop scopes, an APE "knows" that it is inside a loop, and can behave accordingly (e.g., break from the loop).

specification language for abstract programs, syntax and semantics of our logic extensions, including APEs, dynamic frame extensions, and abstract updates, and the calculus rules for SE of APEs. The chapter concludes delineating important aspects of our implementation in KeY.

Chapter 5: Modal and Symbolic Trace Logic Introduces syntax and semantics of Modal Trace Logic (MTL). We characterize MTL, e.g., by applying it to classic axioms of modal and dynamic logic. To demonstrate its usefulness, we formalize a number of program verification techniques in MTL. Finally, we show how to translate MTL to Symbolic Trace Logic (STL) and define a sequent calculus for STL.

Chapter 6: Correctness of Refactoring Techniques We describe how we used Abstract Execution to model and analyze Java refactoring techniques. We present REFINITY, a new frontend for relational proofs of abstract programs. For all analyzed refactorings, we describe constraints distilled as preconditions for behavioral safety of the application of refactoring techniques. The chapter concludes with a discussion of the performance of AE and other aspects related to relational modeling of abstract programs.

Chapter 7: Related Work Figure 1.2 is used to structure our discussion of related work: We assign research areas to each of the nodes in the graph, and discuss related work for (combinations of) these nodes.

Chapter 8: Future Work There are many ways to connect to this thesis, which are outlined in this chapter. We discuss extensions for (applications) of Abstract Execution, MTL/STL, and Symbolic Execution. The detailed record of suggested applications of AE (e.g., abstract cost analysis and Correctness-by-Construction) comprises three projects for which research collaborations are envisaged or have already begun.

Chapter 9: Conclusion This chapter summarizes and concludes the thesis.

In the appendices, we present empirical results of the performance evaluation of the loop scope invariant rule and show the taclet code of our re-implementation of the rule (Appendix A), provide proofs of the SE rules for APEs (Appendix B), display the KeY taclets representing these rules (Appendix C), show MTL/STL-related proofs (Appendix D) and present abstract program models for considered refactoring techniques (Appendix E).

This thesis is structured like a classical concert: It starts with a symphony consisting of several movements (theoretical frameworks) and concludes with something light and cheerful (application to refactoring techniques). Below, we provide some guidelines on how to read only parts of it without compromising understanding.

How to Read This Thesis This thesis contains four *relatively* independent chapters with major contributions. The reader may choose to pick only one or some of them if not interested in all topics. For all contributions, certain knowledge about JavaDL is required; we recommend quickly skimming Sect. 2.2 and coming back whenever something is unclear. Loop scopes (Sect. 2.3) are mentioned at several places, but deeper understanding should not be required (we anyway recommend reading that section, as loop treatment is crucial in SE). Chapter 3 presents our universal, theoretic framework for SE, including a discussion of state merging. It can be read without additional dependencies. Chapter 4 contains our main contribution: Abstract Execution, a framework and implemented reasoning system for automatic analysis of behavioral properties of abstract programs. The semantics of APEs is defined using completion scopes (Sect. 2.4). For illustrative purposes only (precise understanding is not required), we use second-order JavaDL (Sect. 2.5) and Symbolic Execution States (SESs) (Sect. 3.1). Chapter 5 introduces the trace-based semantic framework MTL, as well as STL, a logic for reasoning about symbolic trace inclusions. It uses SESs (Sect. 3.1) and a little bit of AE (for the application to formalizing refactoring tasks), for which we recommend reading the section on the syntax (and intuition) of abstract programs (Sect. 4.1). Finally, Chapter 6 presents our case study for AE in which we modeled Java refactoring rules as abstract programs, defined precise safety preconditions and mechanically proved safety. We recommend acquainting oneself with abstract programs by reading Sect. 4.1. Related work (Chapter 7), future work (Chapter 8) and conclusion (Chapter 9) cover all topics.

Possible errata or additions to this thesis will be made available at

`https://www.dominic-steinhoefel.de/thesis/`

2 Theoretical Foundations

This chapter introduces foundational concepts needed in this thesis, beginning with basic definitions and notations (Sect. 2.1). The program logic for the Java programming language used throughout the thesis, *Java Dynamic Logic (JavaDL)*, is described in Sect. 2.2—essentially a digest of the definitions in [Ahr+16]. There, we also introduce some aspects of the Java Modeling Language, a specification language for Java which we extend for AE in Sect. 4.1. The ensuing sections at least partially contain original contributions: In Sect. 2.3, the concept of *loop scopes*, which is essential for our symbolic treatment of loops, is described; we provide a new definition of its semantics. The section also shows the loop invariant rule built on this concept. Loop scopes are generalized to *completion scopes* in Sect. 2.4. These are container statements allowing to react to arbitrary kinds of (abrupt) completion of contained elements. We use completion scopes to define the semantics of APEs in Sect. 4.2. Finally, we specify a (lightweight) extension of JavaDL for quantification over statements and expressions in Sect. 2.5.

2.1 Basic Definitions

We stipulate basic notions about sets/relations/functions, tuples, and substitutions. The natural numbers—naturally including 0—are denoted by $\mathbb{N}$, (mathematical) integers by $\mathbb{Z}$. The absolute of an integer x is denoted by $|x|$.

Sets, Relations, Functions For the empty set we write both $\emptyset$ and $\{\}$. The *complement* $\overline{S}$ of a set S is the set of elements not in S. For a universe U, it holds that $\overline{S} = U \setminus S$ (*set difference*). The *cardinality* $|S| = n$ of S is the number of elements in S. The *power set* 2^S is defined as the set of all subsets of S. If $R \subseteq A \times B$ is a binary relation, we write $dom(R)$ for the *domain* $\{a \in A \mid (a, b) \in R\}$ of R and $rng(R)$ for the *range* $\{b \in B \mid (a, b) \in R\}$ of R. For a *partial function* f from A to B, we write $f : X \rightharpoonup Y$, and $f : X \to Y$ for a total function.

We define the *composition* $f \circ g$ of functions f and g as $(f \circ g)(x) := g(f(x))$. Following [HTK00], we use this order of composition because it can be extended to composition of *relations*: Let P, Q be relations of arities $n+1$ and $m+1$ on a carrier set U. We define

$$P \circ Q := \{(p_1, \dots, p_n, q_1, \dots, q_m) \mid \exists v \in U;\ (p_1, \dots, p_n, v) \in P,\ (v, q_1, \dots, q_m) \in Q\}$$

Interpreting functions as binary relations, we can compose them by relational composition.

Tuples Tuples are elements of finite products $S_1 \times \cdots \times S_n$. We use the notation $\overrightarrow{t}$ for a tuple without mentioning its elements, and write both $s_1, \dots, s_n$ and $(s_1, \dots, s_n)$ for the same tuple, exposing the elements. The product type S^n consists of all tuples of length n, where all elements are from S. We write $\overrightarrow{t}(i)$ for the i-th element of tuple $\overrightarrow{t}$, where i ranges from 1 to the length of $\overrightarrow{t}$. In contrast to tuples, *sequences* may also be infinite. Tuples are finite sequences.

Substitutions In the context of first-order logic, a *substitution* is a partial function θ associating with every variable v a term $\theta(v)$. We write $\theta = [v_1/t_1, \dots, v_n/t_n]$ to denote the substitution defined by $dom(\theta) = \{v_1, \dots, v_n\}$ and $\theta(v_i) = t_i$. We overload θ for terms: $\theta(v) := v$ for all $v \notin dom(\theta)$, and similarly for atoms which are not (logic) variables. We "push down" applications of θ through the term structure, e.g., $\theta(\varphi \wedge \psi) := \theta(\varphi) \wedge \theta(\psi)$. We refer to [Ahr+16, Sect. 2.2] for a more formal account on substitutions.

We write $\theta(t)$ for the application of substitution θ to term t, but use the brackets shorthand in postfix notation: $t[x/t']$ denotes the substitution of t' for x in t. In this thesis, we also use substitutions of *tuples* of values, and (ab)use the notion of substitution in more general contexts, replacing other entities than logic variables.

2.2 Java Dynamic Logic and the Java Modeling Language

Java Dynamic Logic (JavaDL) is a sorted first-order dynamic logic for the Java programming language. It is the program logic of the KeY program verification platform [Ahr+16]. Its main characteristics are the underlying *type hierarchy* comprising, among a fixed set of types always assumed to be present, the class and interface types of the program of interest, and two types of modal operators, *modalities* and *updates*. The modalities are typical for dynamic logic [HTK00]: The *box modality* $[p]\varphi$ expresses that *if* program p terminates, then it terminates in a state in which the postcondition φ holds; the stronger *diamond modality* $\langle p \rangle \varphi$ additionally requires p to terminate. In this thesis, we only consider sequential Java without concurrency, i.e., every terminating program has exactly one final state. JavaDL can be regarded as an extension of Hoare logic [Hoa69]: The box modality formula $Pre \rightarrow [p]Post$ is similar to the Hoare triple $\{Pre\}\, p\, \{Post\}$. However, in contrast to Hoare logic, JavaDL is closed under common logic operators [Ahr+16].

The second category of modal operator in JavaDL, *updates*, also denotes state changes. However, updates are much more restricted than modalities; for instance, an update cannot fail to terminate. Consider, for example, the formula $[\texttt{x=1;}]\texttt{x} \geq 0$ asserting that, after termination of the Java program "`x=1;`", the variable `x` is positive in the resulting state. This formula is equivalent to the following one using an update instead of a modality: $\{\texttt{x} := 1\}\texttt{x} \geq 0$. Its intended meaning is that in all states where `x` attains the value of 1, the formula $\texttt{x} \geq 0$ holds. The expression $\texttt{x} := 1$ is an update which is *applied* to the postcondition by putting it into curly braces. Subsequently, we define core notions of syntax, semantics and sequent calculus of JavaDL, as well as central characteristics of the concept of *taclets*, a domain-specific language to describe JavaDL sequent calculus rules. Finally, we discuss the basics of the Java Modeling Language (JML) needed for understanding this thesis. For full details about JavaDL, taclets, and also JML, we refer to the KeY book [Ahr+16], where the definitions given here originate.

2.2.1 Syntax

Type Hierarchies and Signatures The syntax of JavaDL terms and formulas is based on a type hierarchy $\mathcal{T}$ and signature Σ, which both depend on the program and statements to be proven about it. In the remainder of this and the subsequent sections, we assume a given Java program (a collection of class definitions) *Prg* which can be compiled without errors. A JavaDL *type hierarchy* is any hierarchy $\mathcal{T} = (\mathrm{TSym}, \sqsubseteq)$ consisting of a set of type symbols TSym and a reflexive and transitive subtype relation $\sqsubseteq$ extending the schema shown in Fig. 2.1. Consequently, any type hierarchy contains at least the types *boolean*, *int*, *Null*, and *Object*, as well as the types *LocSet* of *location sets*, *Seq* of finite-length sequences, *Heap* of *heaps*, *Field* of *fields*, and the type *Any* which is a super type of any type with the exception of *Heap* and *Field*. For technical reasons, the hierarchy also includes the bottom type $\perp$ and top type $\top$. The finite-width Java integer types (`byte`, `short`, `int`, etc.) are all mapped to the unbounded JavaDL type *int* of TSym. JavaDL (and KeY) do not support floating point types, which is why they are not included in the hierarchy. For details about treatment of integers, in particular of overflows, see [Ahr+16].

Symbols of JavaDL can either be *rigid* (the interpretation is not changed throughout

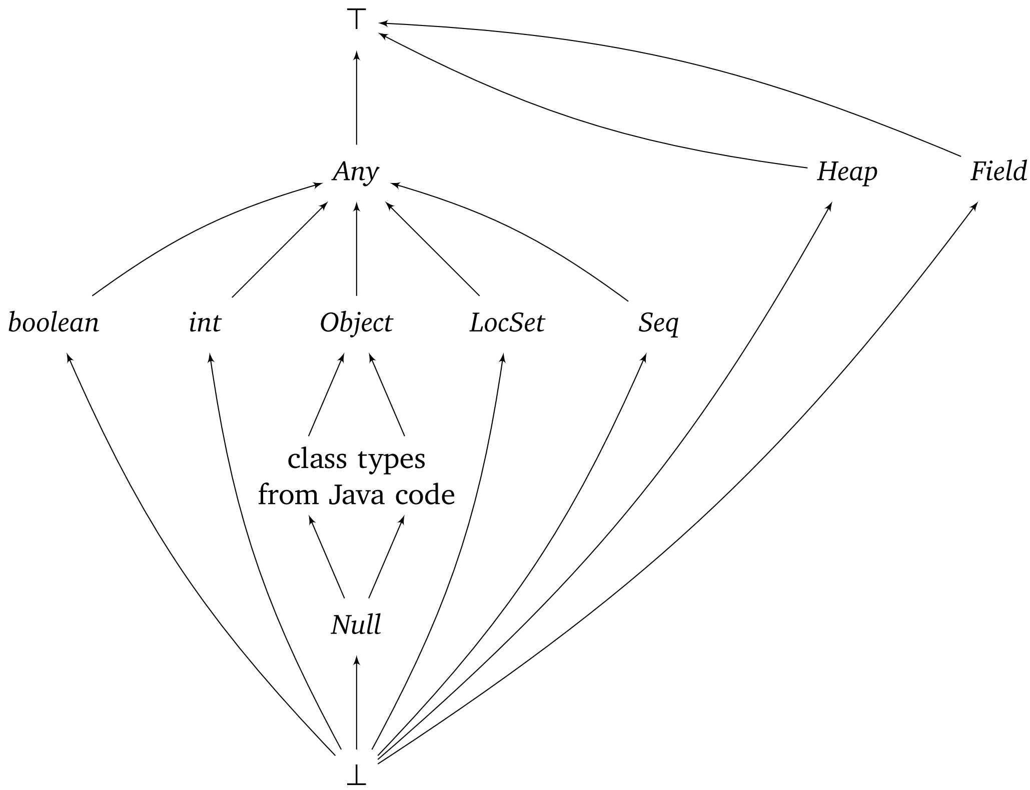


Figure 2.1: Minimal Type Hierarchy for JavaDL

program execution) or *non-rigid* (the interpretation can be changed by the program). *Program variables* are in JavaDL represented by nullary non-rigid function symbols.

int and *boolean*	all function and predicate symbols for *int*, e.g., $+, *, <, \ldots$
	boolean constants *TRUE*, *FALSE*
Java types	$null : Null$
	$length : Object \to int$
	$cast_A : Object \to A$ for any A in $\mathcal{T}$ with $\perp \sqsubset A \sqsubseteq Object$.
	$instance_A : Any \to boolean$ for any type $A \sqsubseteq Any$
	$exactInstance_A : Any \to boolean$ for any type $A \sqsubseteq Any$
Field	$created : Field$
	$arr : int \to Field$
	$f : Field$ for every Java field f
Heap	$select_A : Heap \times Object \times Field \to A$ for any type $A \sqsubseteq Any$
	$store : Heap \times Object \times Field \times Any \to Heap$
	$create : Heap \times Object \to Heap$
	$anon : Heap \times LocSet \times Heap \to Heap$
	$wellFormed(Heap)$
LocSet	$\varepsilon(Object, Field, LocSet)$
	$empty, allLocs : LocSet$
	$singleton : Object \times Field \to LocSet$
	$subset(LocSet, LocSet)$
	$disjoint(LocSet, LocSet)$
	$union, intersect, setMinus : LocSet \times LocSet \to LocSet$
	$allFields : Object \to LocSet,\ allObjects : Field \to LocSet$
	$arrayRange : Object \times int \times int \to LocSet$
	$unusedLocs : Heap \to LocSet$

Figure 2.2: The mandatory vocabulary for JavaDL (Source: [Ahr+16])

Definition 2.1 (JavaDL Signatures)**.** Let $\mathcal{T}$ be a JavaDL type hierarchy for a Java program *Prg*. A JavaDL *signature* w.r.t. $\mathcal{T}$ is a tuple

$$\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym}, \mathrm{PVSym})$$

consisting of sets

- FSym of typed function symbols, where by writing $f : A_1 \times \cdots \times A_n \to A$ we declare the argument types of $f \in \mathrm{FSym}$ to be $A_1, \ldots, A_n$ in the given order and its result type to be A,

- PSym of typed predicate symbols, where by writing $p(A_1, \ldots, A_n)$ we declare the argument types of $p \in \text{PSym}$ to be $A_1, \ldots, A_n$ in the given order; PSym obligatorily contains the equality binary symbol $\doteq (\top, \top)$ as well as the two nullary predicate symbols true and false; the sets FSym and PSym additionally must contain the mandatory vocabulary depicted in Fig. 2.2,
- VSym of typed rigid variable symbols, where by writing $v : A$ for $v \in \text{VSym}$ we declare v to be a variable of type A,
- PVSym of typed non-rigid nullary function symbols called *program variables*, where by writing $\mathtt{x} : A$ for $\mathtt{x} \in \text{PVSym}$ we declare $\mathtt{x}$ to be a program variable of type A. The set PVSym has to contain (1) all local variables declared in *Prg*, (2) an infinite number of symbols for every type, and (3) the distinguished program variable `heap` of type *Heap*. $\Diamond$

We point out that in JavaDL, one can quantify (universally or existentially) over *logical variables* $v \in \text{VSym}$, which however may never occur in programs, while *program variables* $\mathtt{x} \in \text{PVSym}$ may occur in programs, but cannot be quantified over. In this thesis, we use monospaced font for program variables and italic font for logical variables. The meaning of the symbols in Fig. 2.2 is explained whenever one of them occurs in this thesis and is not self-explaining (like the boolean constants TRUE and FALSE); Some essentials about *Heap* and *LocSet* are discussed in Sect. 2.2.6. A complete axiomatization and model-theoretic semantics can be found in [Ahr+16, Sect. 2.4].

Terms and Formulas JavaDL terms and formulas strictly extend First-Order Logic (FOL) terms and formulas by including the modal operators *modalitites* $[p]\varphi$, $\langle p \rangle \varphi$ and *updates*. Updates denote state changes, as do program fragments (in modalities); however, updates always terminate, and expressions in updates never have side effects. We need the notion of *legal program fragments*, which are those fragments that are allowed to appear inside a modality. Simply speaking, a *legal program fragment* p for a *context program Prg* is a sequence of Java statements which may legally appear in the extension of *Prg* by an additional class C with a suitable method m into which p is embedded as a body. By "legally appear" we refer to the rules of the Java Language Specification (JLS) [Gos+05]. Additionally, JavaDL (as implemented in KeY) extends standard Java by certain syntactic categories that are also considered legal. Two important such statement types are *method frames* and *loop scopes*; The latter are explained separately in Sect. 2.3.

Definition 2.2 (Method Frames)**.** A *Method Frame* is a statement of the form

$$\texttt{\textbf{method-frame}}(\texttt{result=}r\texttt{,source=}m(T_1,\ldots,T_n)\texttt{@}T\texttt{,}\texttt{\textbf{this}=}t\texttt{):\{}\ body\ \texttt{\}})$$

where (1) r is a local variable (omitted in case of a void method), (2) $m(T_1,\ldots,T_n)\texttt{@}T$ is a class and method *context*: m is a method of class T with the given signature, (3) t is an expression without side effects and method calls, and (4) *body* is a legal program fragment in the context of *Prg*. The intended semantics of a method frame is that, inside *body* (but outside of nested method frames inside *body*), the visibility rules of the class and method context apply, the Java keyword **`this`** evaluates to the value of t, and the meaning of a **`return`** statement is to assign the returned value to r and to exit the method frame afterward. ◊

Method frames are created from method calls during SE. The idea is to unfold a called method to symbolically execute its context, while at the same time maintaining information about visibility rules and the semantics of **`this`** and **`return`**. For further details about legal program fragments in the context of JavaDL, we refer to [Ahr+16].

We now define the sets Upd of updates, Trm_A of JavaDL terms of type A, and Fml of JavaDL formulas. The definitions of Upd and Trm_A are mutually inductive since they depend on each other. The definitions of terms and formulas are also mutually inductive, which is owed to the *conditional terms*. Those are however syntactic sugar; for every formula with a conditional term there is an equivalent formula without conditional terms.

Definition 2.3 (JavaDL Updates)**.** Let *Prg* be a Java program, $\mathcal{T}$ a type hierarchy for *Prg*, and Σ a gignature for $\mathcal{T}$. The set Upd of updates is inductively defined as:

- $(\mathtt{a} := t) \in \mathrm{Upd}$ for each program variable $\mathtt{a} : A \in \mathrm{PVSym}$ and each term $t \in \mathrm{Trm}_{A'}$ such that $A' \sqsubseteq A$,
- $\mathit{Skip} \in \mathrm{Upd}$,
- $(\mathcal{U}_1 \circ \mathcal{U}_2) \in \mathrm{Upd}$ for all updates $\mathcal{U}_1$, $\mathcal{U}_2 \in \mathrm{Upd}$,
- $(\mathcal{U}_1 \,||\, \mathcal{U}_2) \in \mathrm{Upd}$ for all updates $\mathcal{U}_1$, $\mathcal{U}_2 \in \mathrm{Upd}$,
- $(\{\mathcal{U}_1\}\mathcal{U}_2) \in \mathrm{Upd}$ for all updates $\mathcal{U}_1$, $\mathcal{U}_2 \in \mathrm{Upd}$.

An expression of the form $\{\mathcal{U}\}$, where $\mathcal{U} \in \mathrm{Upd}$, is called an *update application*. ◊

Intuitively, an *elementary update* $\mathtt{a} := t$ assigns the value of term t to program variable $\mathtt{a}$. A *sequential update* $\mathcal{U}_1 \circ \mathcal{U}_2$ denotes the sequential composition of $\mathcal{U}_1$ and $\mathcal{U}_2$: The state changes represented by $\mathcal{U}_1$ are executed after those of $\mathcal{U}_2$. A *parallel update* $\mathcal{U}_1 \,||\, \mathcal{U}_2$

executes the subupdates $\mathcal{U}_1$ and $\mathcal{U}_2$ in parallel. In case of a conflict, i.e., if $\mathcal{U}_1$ and $\mathcal{U}_2$ both assign the same variable, the assignment in $\mathcal{U}_2$ wins. Both sequential and parallel composition are associative, with the *empty update Skip* being their neutral element. The semantics of $\{\mathcal{U}\}$*expr*, i.e., prefixing an expression *expr* (a term, formula, or another update) with an update application, is that *expr* is to be evaluated in the state represented by the update $\mathcal{U}$. The following expressions are equivalent: $\{\mathcal{U}_1\}\{\mathcal{U}_2\}t$, $\{\mathcal{U}_1 \circ \mathcal{U}_2\}t$ and $\{\mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2\}t$. We now define the set of JavaDL terms.

Definition 2.4 (JavaDL Terms)**.** Let $\mathcal{T}$ be a type hierarchy and be Σ a signature for $\mathcal{T}$. The set Trm_A of JavaDL *terms of type A* is, for $A \neq \bot$, inductively defined by

- $v \in \mathrm{Trm}_A$ for each variable symbol $v : A \in \mathrm{VSym}$ of type A,
- $f(t_1,\ldots,t_n) \in \mathrm{Trm}_A$ for each $f : A_1 \times \cdots \times A_n \to A \in \mathrm{FSym}$ and all terms $t_i \in \mathrm{Trm}_{B_i}$ with $B_i \sqsubseteq A_i$ for $1 \leq i \leq n$,
- $(\mathit{if}\,(\varphi)\,\mathit{then}\,(t_1)\,\mathit{else}\,(t_2)) \in \mathrm{Trm}_A$ for $\varphi \in \mathrm{Fml}$ and $t_i \in \mathrm{Trm}_{A_i}$ such that $A_2 \sqsubseteq A_1 = A$ or $A_1 \sqsubseteq A_2 = A$,
- $\{\mathcal{U}\}t \in \mathrm{Trm}_A$ for all updates $\mathcal{U} \in \mathrm{Upd}$ and terms $t \in \mathrm{Trm}_A$. ◊

The terms $\mathit{if}\,(\varphi)\,\mathit{then}\,(t_1)\,\mathit{else}\,(t_2)$ are called *conditional terms*; they are introduced for convenience only. Intuitively, they evaluate to the value of t_1 if φ is true and to the value of t_2 if φ is false. We also similarly introduce conditional formulas in the definition of JavaDL formulas below; a conditional formula $\mathit{if}\,(\varphi)\,\mathit{then}\,(\psi)\,\mathit{else}\,(\xi)$ is equivalent to $(\varphi \land \psi) \lor (\neg\varphi \land \xi)$.

Definition 2.5 (JavaDL Formulas)**.** The set Fml of *formulas* of JavaDL for a given type hierarchy $\mathcal{T}$ and signature Σ is inductively defined as

- $p(t_1,\ldots,t_n) \in \mathrm{Fml}$ for $p(A_1,\ldots,A_n) \in \mathrm{PSym}$, and $t_i \in \mathrm{Trm}_{B_i}$ with $B_i \sqsubseteq A_i$ for all $1 \leq i \leq n$ (those are called *atomic* formulas),
- $(\neg\varphi)$, $(\varphi \land \psi)$, $(\varphi \lor \psi)$, $(\varphi \to \psi)$, $(\varphi \leftrightarrow \psi)$, $(\mathit{if}\,(\varphi)\,\mathit{then}\,(\psi)\,\mathit{else}\,(\xi))$ are in Fml for any φ, ψ, $\xi \in \mathrm{Fml}$,
- $\forall v; \varphi$, $\exists v; \varphi$ are in Fml for $\varphi \in \mathrm{Fml}$ and $v : A \in \mathrm{VSym}$,
- $\langle p \rangle \varphi$, $[p]\varphi$ are in Fml for all legal program fragments p and formulas $\varphi \in \mathrm{Fml}$,
- $\{\mathcal{U}\}\varphi \in \mathrm{Fml}$ for all updates $\mathcal{U} \in \mathrm{Upd}$ and formulas $\varphi \in \mathrm{Fml}$.

We call a formula *closed* if it does not contain free *logic* variables $v \in \mathrm{VSym}$. ◊

2.2.2 Semantics

JavaDL syntax elements are given meaning by so-called *Kripke structures*. We first give the (slightly rephrased) original definition of [Ahr+16]; afterward, we stipulate the notion used in this thesis. In the framework of [Ahr+16], Kripke structures are *collections* of first-order structures. The individual first-order structures within the *same* Kripke structure may assign different values to program variables (the *non-rigid* symbols), but assign the same values to all rigid symbols. Two different Kripke structures may differ on their interpretation of rigid function and predicate symbols.

Definition 2.6 (JavaDL Kripke Structures [Ahr+16])**.** Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg* and Σ a signature w.r.t. $\mathscr{T}$. A *JavaDL Kripke structure* for Σ is a tuple

$$K = (\mathscr{S}, \varrho)$$

consisting of

- an infinite set $\mathscr{S}$ of first-order structures over Σ, called *states*. Each such structure is a tuple $(\mathscr{D}, \delta, I)$ of a set $\mathscr{D}$, the *domain*, a *typing function* $\delta : \mathscr{D} \to \mathrm{TSym} \setminus \{\bot\}$ s.t. for every $A \in \mathrm{TSym}$, the set $\mathscr{D}^A = \{d \in \mathscr{D} | \delta(d) \sqsubseteq A\}$ is not empty, and an *interpretation* I of function and predicate symbols, respecting the typing information, in the usual sense. We demand that any two states $\sigma_1, \sigma_2 \in \mathscr{S}$ coincide in their domain and in the interpretation of predicate and rigid function symbols. Furthermore, $\mathscr{S}$ is closed under this property, i.e., any FOL structure coinciding with the states in $\mathscr{S}$ in the domain and the interpretation of the predicate and function symbols is also in $\mathscr{S}$.
- a function ϱ associating with every legal program fragment p a *transition relation* $\varrho(p) \in \mathscr{S} \times \mathscr{S}$ s.t. $(\sigma_1, \sigma_2) \in \varrho(p)$ iff p, when started in σ_1, terminates normally (without throwing an exception) in σ_2. ◊

We abstain from giving a formal definition of ϱ and instead refer to the JLS [Gos+05] for a description of how to formalize the semantics of Java programs.

The above Def. 2.6 beautifully reuses the well-known concept of first-order structures and introduces little theoretical overhead. If, however, the only *non-rigid* symbols of our logic are program variables (and we do not have, for instance, non-rigid functions of higher arities), we can use an alternative definition. The following definition, which is the one presumed for this thesis, makes the notion of states and the *constant domain assumption* (all first-order structures share the same domain) more explicit. States are *maps from program variables to domain values*. Valuation of program variables is then expressed as $\sigma(\mathsf{x}) \in \mathscr{D}^A$, for a state $\sigma \in \mathscr{S}$ and $\mathsf{x} : A \in \mathrm{PVSym}$.

Definition 2.7 (JavaDL Kripke Structures)**.** Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg* and Σ a signature w.r.t. $\mathscr{T}$. A *JavaDL Kripke structure* for Σ is a tuple

$$K = (\mathscr{D}, \delta, I, \mathscr{S}, \varrho)$$

where

- the *domain* $\mathscr{D}$ and *typing function* δ are as in Def. 2.6,
- I is an *interpretation function* assigning functions to function symbols $f \in \mathrm{FSym}$ and relations to predicate symbols $p \in \mathrm{PSym}$, respecting the typing,
- the set of *states* $\mathscr{S}$ consists of functions $\sigma : \mathrm{PVSym} \to \mathscr{D}$ mapping program variables $\mathsf{x} : A$ to domain values $\sigma(\mathsf{x}) \in \mathscr{D}^A$,
- ϱ is a function mapping legal program fragments to transition relations as in Def. 2.6.

$\Diamond$

We define the semantics of JavaDL expressions by providing a *valuation function* $val(K, \sigma, \beta | expr)$ which assigns to terms a domain value of $\mathscr{D}$, to formulas a truth value *tt* or *ff*, and to updates a state transformer $\mathscr{S} \to \mathscr{S}$. The valuation function depends on (1) a JavaDL Kripke *structure* as in Def. 2.7 which defines the domain and the evaluation of rigid symbols $f \in \mathrm{FSym}$ and $p \in \mathrm{PSym}$ as well as of Java programs, (2) a (Kripke) *state* for the evaluation of non-rigid program variables $\mathsf{x} \in \mathrm{PVSym}$, and (3) a variable assignment function $\beta : \mathrm{VSym} \to \mathscr{D}$ for free logic variables $v \in \mathrm{VSym}$. For formulas, we also write $K, \sigma, \beta \models \varphi$ for $val(K, \sigma, \beta | \varphi) = tt$ and $K, \sigma, \beta \not\models \varphi$ for $val(K, \sigma, \beta | \varphi) = ff$. For closed terms, we omit β.

Definition 2.8 (JavaDL Semantics)**.** Let *Prg* be a Java program, $\mathscr{T}$ be a type hierarchy for *Prg*, Σ a signature w.r.t. $\mathscr{T}$, $K = (\mathscr{D}, \delta, I, \mathscr{S}, \varrho)$ a Kripke structure for Σ, $\sigma \in \mathscr{S}$ a state, and $\beta : \mathrm{VSym} \to \mathscr{D}$ a variable assignment. The evaluation $val(K, \sigma, \beta | \cdot)$ of JavaDL updates, terms and formulas is defined as in Figure 2.3. In the figure, we omit the definitions for propositional junctors and first-order quantifiers and instead refer to [Ahr+16]. $\Diamond$

For the semantics of the mandatory vocabulary of JavaDL, we refer to [Ahr+16, Fig. 2.11]. Some essential concepts of the semantics for the *Heap* and *LocSet* functions are discussed later in this section in Sect. 2.2.6.

We consider deterministic Java programs: There is at most one σ' with $(\sigma, \sigma') \in \varrho(p)$ for each $\sigma \in \mathscr{S}$. For sets of formulas $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_n\} \subseteq \mathrm{Fml}$, $n \geq 0$, we define

Updates

$$
\begin{array}{rcl}
val(K,\sigma,\beta|\cdot) & : & \mathrm{Upd} \to (\mathscr{S} \to \mathscr{S}) \\
val(K,\sigma,\beta|\mathsf{x} := t)(\sigma')(\mathsf{y}) & = & \begin{cases} val(K,\sigma,\beta|t) & \text{if } \mathsf{y} = \mathsf{x} \\ \sigma'(\mathsf{y}) & \text{otherwise} \end{cases} \\
val(K,\sigma,\beta|Skip)(\sigma') & = & \sigma' \\
val(K,\sigma,\beta|(\mathscr{U}_1 \,||\, \mathscr{U}_2))(\sigma') & = & val(K,\sigma,\beta|\mathscr{U}_2)(val(K,\sigma,\beta|\mathscr{U}_1)(\sigma')) \\
val(K,\sigma,\beta|\{\mathscr{U}_1\}\mathscr{U}_2) & = & val(K,val(K,\sigma,\beta|\mathscr{U}_1)(\sigma),\beta|\mathscr{U}_2) \\
val(K,\sigma,\beta|\mathscr{U}_1 \circ \mathscr{U}_2) & = & val(K,\sigma,\beta|(\mathscr{U}_1 \,||\, \{\mathscr{U}_1\}\mathscr{U}_2))
\end{array}
$$

Terms

$$
\begin{array}{rcl}
val(K,\sigma,\beta|\cdot) & : & \mathrm{Trm}_A \to \mathscr{D}^A \\
val(K,\sigma,\beta|v) & = & \beta(v) \\
val(K,\sigma,\beta|\mathsf{x}) & = & \sigma(\mathsf{x}) \\
val(K,\sigma,\beta|f(t_1,\ldots,t_n)) & = & I(f)(val(K,\sigma,\beta|t_1),\ldots,val(K,\sigma,\beta|t_n)) \\
val(K,\sigma,\beta|if(\varphi)\,then(t_1)\,else(t_2)) & = & \begin{cases} val(K,\sigma,\beta|t_1) & \text{if } val(K,\sigma,\beta|\varphi) = tt \\ val(K,\sigma,\beta|t_2) & \text{if } val(K,\sigma,\beta|\varphi) = f\!f \end{cases} \\
val(K,\sigma,\beta|\{\mathscr{U}\}t) & = & val(K,val(K,\sigma,\beta|\mathscr{U})(\sigma),\beta|t)
\end{array}
$$

Formulas

$$
\begin{array}{rcl}
val(K,\sigma,\beta|\cdot) & : & \mathrm{Fml} \to \{tt, f\!f\} \\
val(K,\sigma,\beta|\,\mathrm{true}) & = & tt \\
val(K,\sigma,\beta|\,\mathrm{false}) & = & f\!f \\
val(K,\sigma,\beta|\neg\varphi) & = & \begin{cases} tt & val(K,\sigma,\beta|\varphi) = f\!f \\ f\!f & val(K,\sigma,\beta|\varphi) = tt \end{cases} \\
 & \vdots & \\
val(K,\sigma,\beta|\{\mathscr{U}\}\varphi) & = & val(K,val(K,\sigma,\beta|\mathscr{U})(\sigma),\beta|\varphi) \\
val(K,\sigma,\beta|[p]\varphi) & = & \begin{cases} tt & \text{if there is no } \sigma' \text{ with } (\sigma,\sigma') \in \varrho(p) \text{ or} \\ & val\big(K,\sigma',\beta|\varphi\big) = tt \text{ for } \sigma' \text{ with } (\sigma,\sigma') \in \varrho(p) \\ f\!f & \text{otherwise} \end{cases} \\
val(K,\sigma,\beta|\langle p\rangle\varphi) & = & \begin{cases} tt & \text{if there is a } \sigma' \text{ with } (\sigma,\sigma') \in \varrho(p) \text{ and} \\ & val\big(K,\sigma',\beta|\varphi\big) = tt \text{ for } \sigma' \text{ with } (\sigma,\sigma') \in \varrho(p) \\ f\!f & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2.3: JavaDL Semantics

$\bigwedge \Phi :=$ true $\wedge \varphi_1 \wedge \cdots \wedge \varphi_n$ and $\bigvee \Phi :=$ false $\vee \varphi_1 \vee \cdots \vee \varphi_n$. Subsequently, we define the usual notions of *satisfiability* and *validity* of JavaDL formulas.

Definition 2.9 (Satisfiability and Validity)**.** Let *Prg* be a Java program, $\mathcal{T}$ be a type hierarchy for *Prg*, Σ a signature w.r.t. $\mathcal{T}$, and $\varphi \in$ Fml a formula. We call φ *satisfiable* if there is a Kripke structure $K = (\mathcal{D}, \delta, I, \mathcal{S}, \varrho)$, a state $\sigma \in \mathcal{S}$ and a variable assignment β such that $K, \sigma, \beta \models \varphi$. The formula φ is called *(logically) valid*, denoted by $\models \varphi$, if $K, \sigma, \beta \models \varphi$ for all Kripke structures $K = (\mathcal{D}, \delta, I, \mathcal{S}, \varrho)$, all states $\sigma \in \mathcal{S}$, and all variable assignments β. ◊

2.2.3 Update Simplification Rules

$$\{\ldots \,||\, \mathsf{a} := t_1 \,||\, \ldots \,||\, \mathsf{a} := t_2 \,||\, \ldots\}t \leadsto \{\ldots \,||\, \mathit{Skip} \,||\, \ldots \,||\, \mathsf{a} := t_2 \,||\, \ldots\}t \quad \text{where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \qquad \mathsf{dropUpdate}_1$$

$$\{\ldots \,||\, \mathsf{a} := t' \,||\, \ldots\}t \leadsto \{\ldots \,||\, \mathit{Skip} \,||\, \ldots\}t \quad \text{where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd},\ \mathsf{a} \notin \mathit{fpv}(t) \qquad \mathsf{dropUpdate}_2$$

$$\{\mathcal{U}\}\{\mathcal{U}'\}t \leadsto \{\mathcal{U} \,||\, \{\mathcal{U}\}\mathcal{U}'\}t \quad \text{where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \qquad \mathsf{seqToPar}$$

$$\{\mathcal{U}\}f(t_1, \ldots, t_n) \leadsto f(\{\mathcal{U}\}t_1, \ldots, \{\mathcal{U}\}t_n) \quad \text{where } f \in \mathrm{FSym} \cup \mathrm{PSym} \qquad \mathsf{applyOnRigid}_2$$

$$\{\mathcal{U}\}\mathsf{a} := t \leadsto \mathsf{a} := \{\mathcal{U}\}t \qquad \mathsf{applyOnRigid}_7$$

$$\{\mathcal{U}\}(\mathcal{U}_1 \,||\, \mathcal{U}_2) \leadsto (\{\mathcal{U}\}\mathcal{U}_1 \,||\, \{\mathcal{U}\}\mathcal{U}_2) \qquad \mathsf{applyOnRigid}_8$$

$$\{\mathsf{a} := t\}\mathsf{a} \leadsto t \qquad \mathsf{applyOnTarget}$$

$$\vdots$$

Figure 2.4: Update Simplification Rules (Excerpt from [Ahr+16, Table 3.1])

Some of the most frequently applied rules in JavaDL proofs simplify updates generated during symbolic execution of Java programs. Figure 2.4 shows an excerpt of the most relevant such simplification schemes; we refer to [Ahr+16, Sect. 3.4.2] for a full account.

Rule $\mathsf{dropUpdate}_1$ formalizes the intuition that within a parallel update, the "last one wins". The $\mathsf{dropUpdate}_2$ rule allows dropping an *ineffective* elementary update that assigns a variable which does not occur freely in the target. A *free* occurrence of a program variable is any occurrence, except for one inside a program fragment which is bound by a declaration within that fragment. Furthermore, we always assume that program fragments have an (implicit) free occurrence of the program variable heap. The function $\mathit{fpv} : \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \rightarrow 2^{\mathrm{PVSym}}$ is defined according to this intuition.

The rule seqToPar converts a sequential update cascade to a single update. Due to the "last one wins" semantics for parallel updates, this is done by applying the first update to the second, and replacing sequential by parallel composition.

We only show one of the rules, $\mathsf{applyOnRigid}_2$, for applying updates to (rigid) operators. All rules of this class propagate the update to the subterms below the operator. Ultimately, an update can either be simplified away by one of the drop rules, or applied to the target variable itself by the applyOnTarget rule.

2.2.4 Calculus

The calculus of JavaDL is a *sequent calculus* [Gen35]. A *proof* using the calculus is a tree where the nodes are *sequents* of the form $\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$. The formulas $\varphi_1, \ldots, \varphi_n$ at the left-hand side of the sequent separator $\vdash$ are the *antecedents* of the sequent (all of them together form the *antecedent*); the formulas $\psi_1, \ldots, \psi_m$ are the *succedents* (all of them together form the *succedent*). Both antecedent and succedent are sets, i.e., the order and multiple occurrences are irrelevant. A sequent $\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$ is *valid* iff the formula $\bigwedge_{i=1}^{n} \varphi_i \rightarrow \bigvee_{j=1}^{m} \psi_j$ is valid.

A sequent calculus proof tree is constructed starting from a root sequent by repeatedly applying sequent calculus *rules* to the leaves of the tree. Such a rule has the following form:

$$\mathsf{ruleName}\ \frac{P_1, \ldots, P_n}{C}$$

The sequents $P_1, \ldots, P_n$ are called the *premises* and the sequent C the *conclusion* of the rule. We use schematic variables Γ, Δ for sets of formulas and the comma notation Γ, φ for $\Gamma \cup \{\varphi\}$. Further frequently used schematic variables are ψ, φ for formulas, t, c for terms and constants, v, w for logic variables and x, y for program variables. An instance of a rule is obtained by consistently replacing the schematic variables in premise and conclusion by corresponding instantiations. Rules are applied bottom-up: On a proof tree leaf with sequent s we can apply a rule where the conclusion can be instantiated to s. Afterward, the tree contains as new leaves the corresponding instantiations of the rule's premises. Rules with zero premises are called *closing rules*. A branch in a proof tree is called *closed* its last rule application is a closing rule. A proof tree is called *closed* if all of its branches are closed. We say that a sequent $\Gamma \vdash \Delta$ can be *derived* if there is a closed proof tree which has $\Gamma \vdash \Delta$ as root.

The JavaDL calculus comprises rules for first-order reasoning, equality rules, and rules for reasoning about theories like integers and finite sequents. For details about these rules, we refer to the KeY book [Ahr+16] or to text books about logics, like [Gal86]. Additionally, JavaDL contains rules for update simplification and for *Symbolic Execution (SE)*. SE rules operate on the *first "active" statement* p in a modality $[\pi\ p\ \omega]$ or $\langle \pi\ p\ \omega \rangle$. The *nonactive prefix* π consists of opening braces, labels, beginnings "**try** { " of try-catch-finally blocks, and beginnings of method frames and loop scopes. The purpose of the prefix is to correctly resolve field and method bindings, and to keep track of the blocks of which p is part of; this is needed to appropriately handle statements like **throw** and **return**. The *postfix* ω contains remaining statements and closing braces, endings of method frames and loop scopes, etc. The program $\pi\ p\ \omega$ always has to be a legal program fragment.

Example 2.1 (Symbolic Execution Rules). Consider the following modality, where the active statement "i=0;" is wrapped in a labeled **try**-**finally** block, and the nonactive prefix π and the "rest" ω are the indicated parts of the program:

$$[\underbrace{\texttt{l:\{}\mathbf{try}\texttt{ \{}}_{\pi}\ \underbrace{\texttt{i = 0;}}_{p}\ \underbrace{\texttt{j = 0;\} }\mathbf{finally}\texttt{ \{k = 0;\}\}}}_{\omega}](\texttt{i} \doteq 0)$$

The sequent $\texttt{i} < 0 \vdash [\pi\ \texttt{i=0;}\ \omega](\texttt{i} \doteq 0)$, embedding this modality, intuitively expresses "when started in a state where i is negative, "π i=0; ω" either does not terminate, or terminates in a state where i is zero (since Java is deterministic)". The SE rule applicable to the sequent, assignment, transforms the active statement into an update. Below, we show the definition of this rule on the left and its instantiation for the sequent on the right:

$$\left[\mathsf{assignment}\ \frac{\Gamma \vdash \{\texttt{x} := expr\}[\pi\ \omega]\varphi, \Delta}{\Gamma \vdash [\pi\ \texttt{x=}expr\texttt{;}\ \omega]\varphi, \Delta}\right] \qquad \frac{\texttt{i} < 0 \vdash \{\texttt{i} := 0\}[\pi\ \omega](\texttt{i} \doteq 0)}{\texttt{i} < 0 \vdash [\pi\ \texttt{i=0;}\ \omega](\texttt{i} \doteq 0)}$$

SE rules can also have more than one premise. The following rule can be used to symbolically execute an **`if`** statement:

$$\text{ifElseSplit}\ \frac{\Gamma, se \doteq \mathrm{TRUE} \vdash [\pi\ p_1\ \omega]\varphi, \Delta \quad \Gamma, se \doteq \mathrm{FALSE} \vdash [\pi\ p_2\ \omega]\varphi, \Delta}{\Gamma \vdash [\pi\ \mathbf{if}\ (se)\ p_1\ \mathbf{else}\ p_2\ \omega]\varphi, \Delta}$$

The abbreviation *se* stands for "simple expression" and represents Java expressions without side effects. The expression `i++`, for example, is *not* a simple expression. ◊

The above example demonstrates basic ideas of SE (in JavaDL): Programs inside modalities are transformed step-by-step to updates which are, after execution, applied to their target formulas. Thus, the original problem is reduced to first-order reasoning. Whenever the execution depends on the concrete value of an expression, as in rule ifElse, it splits into several branches that are followed independently. This also leads to the so-called *state explosion problem* of SE which we discuss in more detail in Sect. 3.3.

Note that rule assignment above would also have been applicable if there was an additional update in front of the transformed sequent; this update would then appear both in the premise and conclusion.

For more details like the simplification rules for updates or additional examples of SE rules, we again refer to [Ahr+16].

Soundness and Completeness The most important property of any validity calculus is *soundness*, the property that only valid formulas are derivable. Soundness for the calculus, as formalized in the proposition below, follows from the soundness of all of its *rules*: If the premises of a rule are valid, then the conclusion also has to be valid.

Proposition 2.1 (Soundness of the JavaDL Calculus). *If a sequent $\Gamma \vdash \Delta$ is derivable in the JavaDL calculus, then it is valid, i.e., the formula $\bigwedge \Gamma \to \bigvee \Delta$ is logically valid.*

Completeness is the property that all valid sequents, which comprises all true statements about programs in the case of JavaDL, can also be derived. This is not possible in JavaDL, first because it comprises first-order arithmetic and is therefore incomplete by Gödel's Incompleteness Theorem [Göd31]; and second, a complete JavaDL calculus would yield a decision procedure for the undecidable Halting Problem. What nevertheless *can* be stated for the calculus is a notion of *relative completeness*, a completeness "up to" the inherent incompleteness in the first-order part. The idea is that a relatively complete calculus contains all the rules that are necessary to prove valid program properties; it may only fail to do so if the proof required the derivation of a nonprovable first-order property (such as an instance of a higher-order induction axiom). The following proposition from [Ahr+16] is discussed in more detail there.

Proposition 2.2 (Relative Completeness of the JavaDL Calculus)**.** *If a sequent* $\Gamma \vdash \Delta$ *is valid, i.e., the formula* $\bigwedge \Gamma \rightarrow \bigvee \Delta$ *is logically valid, then there is a finite set* Γ_{FOL} *of logically valid first-order formulas such that the sequent*

$$\Gamma_{FOL}, \Gamma \vdash \Delta$$

is derivable in the JavaDL sequent calculus.

A useful property of sequent calculi is that a calculus is sound if, *and only if*, all of its *rules* are sound (see, e.g., [Ahr+16, Lemma 4.7]). There is a more complicated relation between completeness of all rules and completeness of the calculus (informally, there have to be complete elimination rules for all connectives, plus closing rules for atomic propositions); a complete calculus can have *in*complete rules. It suffices therefore to locally show soundness, and, under restrictions, completeness *for all rules* of the system to derive the corresponding global property. This is especially useful when adding rules to an existing (sound and/or (relatively) complete) system, such as our Abstract Execution rules in Chapter 4. We define soundness and completeness of sequent calculus rules.

Definition 2.10 (Soundness and Completeness of Calculus Rules [Ahr+16, Def. 2.21])**.** A rule

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2}{\Gamma \vdash \Delta}$$

of a sequent calculus is called

- *sound* if, whenever $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$ are universally valid, so is $\Gamma \vdash \Delta$.
- *complete* if, whenever $\Gamma \vdash \Delta$ is universally valid, then also $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$ are universally valid.

For rules with a different number of preconditions (non-branching or more than two branches) and with side conditions, the requirements have to be modified accordingly. ◊

2.2.5 Method and Loop Specifications

As deductive program verification so far does not scale to complete software libraries,[1] it is crucial to *modularize* the verification process. This enables asserting strong guarantees about chosen, critical routines (see, e.g., [GBR14; Bec+17; Gou+19]). Key tools in modular verification of Java programs are *method contracts* and *loop specifications*, which allow to treat methods and loops not by inlining and unwinding, but by a calling them "by contract" and using a loop invariant rule. In Sect. 2.3, we discuss in some detail a loop invariant rule making use of loop specifications. Here, we show the definitions for the specification constructs of method contracts and loop specifications. For a method contract *rule* and complete details, we refer to [Ahr+16, Sect. 3.7].

Definition 2.11 (Functional Method Contracts)**.** A *functional JavaDL method contract* for a method or constructor

$$R\ \mathtt{m}(T_1\ \mathtt{p}_1,\ \ldots,\ T_n\ \mathtt{p}_n)$$

declared in class C is a quadruple

$$(pre, post, mod, term)$$

that consists of

- a precondition $pre \in \mathrm{Fml}$,
- a postcondition $post \in \mathrm{Fml}$,
- a modifier set $mod \in \mathrm{Trm}_{LocSet} \cup \{\textsc{strictlyNothing}\}$, and
- a termination witness $term \in \mathrm{Trm}_{Any} \cup \{\textsc{partial}\}$.

Contract components may contain special program variables referring to the execution context:

- $\mathtt{self} : C$ for references to the receiver object (not available if $\mathtt{m}$ is a static method),
- $\mathtt{p}_1 : T_1, \ldots, \mathtt{p}_n : T_n$ representing the method's formal parameters,
- $\mathtt{heap} : Heap$ to access heap locations,
- $\mathtt{heap}^{pre} : Heap$ to access heap locations in the state in which the operation was invoked (in the postcondition only),
- $\mathtt{exc}$: Exception to refer to the exception in case that the method completes abruptly due to a thrown exception (in the postcondition only),

[1] See Chapter 3 for a discussion of scalable SE systems for proving "lightweight" properties vs. "heavyweight" systems (including JavaDL/KeY) which are less scalable, but can prove complex functional properties.

- `res` : R to refer to the result value of a method with a return type different from void (in the postcondition only). ◊

In the above definition, the symbol STRICTLYNOTHING is an indicator subject to special treatment when proving and applying the method contract, and is *not* itself a term. The symbol PARTIAL indicates that the contract is partial and does not require the method to terminate (this is achieved by specifying the method with **`\diverges true;`** in JML).

Loop specifications also have modifier sets and termination witnesses, but instead of a pre- and postcondition, they contain a *loop invariant*. The latter can be regarded as both a pre- and postcondition of the loop body. Loop invariants only have to hold at each *entry point* of the loop; They do not have to be shown after abrupt completion of the loop body (e.g., due to a **`break`** or **`return`**). Loop specifications are defined as follows.

Definition 2.12 (Loop Specifications). A *loop specification* is a triple

$$(inv, mod, term)$$

consisting of

- a loop invariant $inv \in \mathrm{Fml}$,
- a modifier set $mod \in \mathrm{Trm}_{LocSet} \cup \{\textsc{strictlyNothing}\}$,
- a termination witness $term \in \mathrm{Trm}_{Any} \cup \{\textsc{partial}\}$.

Specification components may contain the following program variables referring to the execution context:

- all local variables that are defined in the context of the loop,
- `self` : C for references to the receiver object of the current method frame (not available if the frame belongs to a static method),
- `heap` : *Heap* referring to the heap in the state after the current iteration,
- $\mathtt{heap}^{pre}$: *Heap* referring to the heap in the initial state of the immediately enclosing method frame. ◊

2.2.6 Heap Model

The state of a Java program is determined by the values of the local program variables and the *heap*. A heap is a collection of mappings from pairs of objects and fields to values. The heap model of JavaDL follows the *theory of arrays* [McC63]. Its realization for JavaDL is first described in [SUW11] and also discussed in great detail in [Ahr+16]. Fields of Java programs are represented by the type *Field* and heaps by type *Heap*; both types are obligatory types in any JavaDL type hierarchy $\mathcal{T}$. Reading and writing to a heap is accomplished via function symbols $select_A : Heap \times Object \times Field \to A$, where $A \in \mathcal{T}$, and $store : Heap \times Object \times Field \times Any \to Heap$. Functions which take as first argument objects of type *Heap*, like the family of functions $select_A$, are called *observer functions*. Figure 2.5 shows some selected rules for the heap theory of JavaDL.

selectOfStore	$select_A(store(h, o, f, x), o_2, f_2) \rightsquigarrow$ $\quad if\,(o \doteq o_2 \land f \doteq f_2 \land f \not\doteq created)$ $\quad\quad then\,(cast_A(x))\,else\,(select_A(h, o_2, f_2))$
selectOfCreate	$select_A(create(h, o), o_2, f) \rightsquigarrow$ $\quad if\,(o \doteq o_2 \land o \not\doteq null \land f \doteq created)$ $\quad\quad then\,(cast_A(\mathrm{TRUE}))\,else\,(select_A(h, o_2, f))$
selectOfAnon	$select_A(anon(h, s, h'), o, f) \rightsquigarrow$ $\quad if\,((\varepsilon(o, f, s) \land f \not\doteq created) \lor \varepsilon(o, f, unusedLocs(h)))$ $\quad\quad then\,(select_A(h', o, f))\,else\,(select_A(h, o, f))$
cast	$cast_A(t) \rightsquigarrow t \qquad$ for $t \in \mathrm{Trm}_{A'}$ and $A' \sqsubseteq A$

where o, o_2 : *Object*, f, f_2 : *Field*, h, h' : *Heap*, s : *LocSet*, and $\varepsilon : Object \times Field \times LocSet$ is the "is-element-of" predicate for location sets of type *LocSet*.

Figure 2.5: Some Rewrite Rules for the Heap Theory of JavaDL

The observer functions $select_A$ and $cast_A$ are needed because JavaDL is not dependently typed. The implicit field *created* is used to model that an object has been created on a heap. The function $anon : Heap \times LocSet \times Heap \to Heap$ is used to anonymize the fields of the location set in the first heap; when accessing those, the second heap is used. This function is usually used to anonymize heap elements that are assigned in the body of a loop when applying a loop invariant rule. Then, the second argument heap is a fresh symbol, which explains the term "anonymization".

JavaDL's heap concept is an overapproximation and allows to express situations which cannot occur in practice. To establish certain "wellformedness conditions", the predicate *wellFormed*(`heap`) has been included in the mandatory vocabulary. It is axiomatized on a pragmatic basis, not preventing *all* practically impossible situations. For instance, it is not guaranteed that, when selecting a field from an object, that this field actually exists in the object's class. We name one axiom of *wellFormed* and, as usual, refer to [Ahr+16] for details: The rule onlyCreatedObjectsAreReferenced asserts that on each wellformed heap, the value of a field f is either *null* or refers to an object that has been created:

$$wellFormed(h) \rightarrow select_A(h,o,f) \doteq null \lor select_{boolean}(h, select_A(h,o,f), created) \doteq \mathrm{TRUE}$$

Semantics We explain some details of the semantics of the *Heap* and related *LocSet* types. The domain of *Heap* are all functions $o \rightarrow f \rightarrow val$ from objects and fields to domain values. Heap store expressions $store(h,o,f,x)$ update function h at the given position (o,f) with value x. The semantics of $anon(h_1,s,h_2)$ is to update heap h_1 such that for all locations in location set s and those not occurring in h_1, the value in h_2 is returned; otherwise, the value in h_1 is selected. For modeling arrays, the *arr* function is an injective function from $\mathbb{Z}$ into the *Field* domain. Inside a concrete state $\sigma \in \mathcal{S}$, the heap is represented at the position of the special program variable `heap` : *Heap*. For instance, the value of a location (o,f), for an object o and field f, in state σ can be retrieved by $\sigma(\texttt{heap})(o,f)$. The domain of the type *LocSet* are pairs of objects and fields. In Sect. 4.2, we extend this by *program variable locations*. For a full account, we refer to [Ahr+16, Sect. 2.4].

Remark 2.1 (Simplified Syntax for State Access). To increase readability, we frequently use a simplified syntax to retrieve the value of a location in a state. Let *loc* be a location (either a program variable or an object-field pair), and $\sigma \in \mathcal{S}$. Then, we write $\sigma(loc)$ for

- $\sigma(loc)$ if *loc* is a program variable, and
- $\sigma(\texttt{heap})(o,f)$ if *loc* is an object-field pair (o,f). ◊

2.2.7 Taclets

KeY's *taclet* language is a domain-specific language for sequent calculus rules. Taclets are comparable in expressivity, but more formal, than rules in textbook notation. The language is general enough to cover all rules of a first-order sequent calculus and most rules of calculi for dynamic logic [Ahr+16]. We point out that it is rather easy to extend the taclet mechanism by "hooking in" new conditions and transformers implemented as Java code. At the time of writing this thesis, some rules (like complex rules for method call treatment) are still implemented as "built-in" Java rules and not as taclets. We have shown that also complex rules can be realized as taclets by writing taclets for both *loop invariant* and *abstract execution rules* after suitable additions of conditions and transformers. This is discussed in more detail in Sects. 2.3 and 4.4.

Taclets describe (1) to which part of a sequent (2) under which conditions the taclet can be applied, and (3) in which way the sequent is modified yielding new proof goals.

They are declared inside KeY input files (text files with ending ".key"); their syntax is part of KeY's input syntax. Taclets in ".key" files are contained in a `\rules { ... }` section following the grammar in Fig. 2.7 (Page 36). The clause for the nonterminal "*variableCondition*" for *variable conditions* (conditions defining the applicability of a taclet) is abbreviated, since there are many variable conditions. Some nonterminals, in particular *schematicSequent*, *schematicFormula* and *schematicTerm*, are not further expanded; We refer to [Ahr+16, Appendix B] for a formal definition of their basic syntax. In principle, they are sequents, formulas and terms containing schematic elements, in particular *schema variables* and *meta constructs*. Schema variables are instantiated when a taclet is matched during proof construction. Meta constructs comprise *term transformers* and *program transformers*. Those constitute another powerful mechanism for extending the expressivity of the taclet language. They take as arguments a (potentially schematic) term or program and return a transformed term or program obtained by calling a Java method.

We conclude this section with an example. For a more complete reference about the taclet mechanism, we refer the curious reader to [Ahr+16, Chapter 4]. Examples of more complicated taclets are given and explained in Sects. 2.3 and 4.4 of this thesis.

Listing 2.1: Taclet: `pullOut`

```
pullOut {
  \schemaVar \term int t;
  \schemaVar \skolemTerm int sk;
  \find ( t )
  \sameUpdateLevel
  \replacewith ( sk )
  \add ( t = sk ==> )
}
```

Listing 2.2: Taclet: `ifElseSplit`

```
ifElseSplit {
  \find (==> \modality{#allmodal}{
      .. if(#se) #s0 else #s1 ...
    }\endmodality(post))
  "if_#se_true":
    \replacewith (==> \modality{#allmodal}{
        .. #s0 ...
      }\endmodality(post))
    \add (#se = TRUE ==>);
  "if_#se_false":
    \replacewith (==> \modality{#allmodal}{
        .. #s1 ...
      }\endmodality(post))
    \add (#se = FALSE ==>)

  \heuristics(split_if)
  \displayname "ifElseSplit"
};
```

Figure 2.6: Example Taclets

Example 2.2 (Example Taclets)**.** Figure 2.6 shows taclets `pullOut` and `ifElseSplit`. The former "pulls out" a term from anywhere in a sequent by replacing it with a Skolem term ("`\replacewith (sk)`") and adding an antecedent formula stating that the Skolem term equals the pulled out term ("`\add ( t = sk ==> )`"). The symbol "`==>`" represents the sequent separator "$\vdash$"; the fact that the new formula appears on its left signifies that it is an antecedent formula. The directive "`\sameUpdateLevel`" sets the taclet to a different "mode": All updates occurring in front of the matched term `t` are also added in front `\assumes`, `\add` and `\replacewith` formulas, the latter two of which apply in this case. If this mode switch was omitted, any updates in front of the pulled out term `t` would not occur in front of the new antecedent formula (but they would still be present after the *replacement* at the original position).

The taclet `ifElseSplit` corresponds to the rule ifElseSplit shown in text book style in Example 2.1 (Page 27). It is a *symbolic execution taclet* and additionally showcases some further features of the taclet language. First, schema variables in programs are prefixed by the "#" symbol in taclet syntax. Programs occur in modalities, which are declared using the syntax `\modality{` *modalityType*`}{` *prog* `}\endmodality(`*postCond*`)`. In the example, *modalityType* is a schema variable `#allmodal` that has been declared to match any modality. The postcondition *postCond* is a schema variable `post` of type `\formula`. Inside a schematic program, the nonactive prefix π is denoted with two dots "`..`" and the postfix ω with three dots "`...`". The taclet produces two proof branches; this is accomplished by separating more than one goal template with "`;`". In the example, the branches are given names (the strings before the colons). The KeY user interface replaces schema variables in these names by their actual instantiations when displaying proof trees. The line "`\heuristics(split_if)`" adds the taclet to the *rule set* "split_if". The primary use case of rule sets is to improve the strategies used in automatic rule application. For example, the rule set "split_if" could receive a very low priority since it splits the proof (and can thus lead to unnecessary state explosion).[2] Finally, one can give taclets user-friendly names with the "`\displayname`" keyword. In the user interface, those names will be shown instead of actual taclet names. They need not be unique: several rules can have the same display name. In the example of `ifElseSplit`, the display name actually equals the real name. The reason for this (technically superfluous) specification is that there is another taclet, `ifSplit`, matching **`if`** statements without an **`else`** branch. Both rules have the same display name. For documentary reasons, the display name is also provided in our example rule. ◊

[2] Instead, one could choose to prefer more efficient rules using conditional formulas. This delays, and possibly prevents, proof splits.

Remark. In the above Example 2.2, we used typewriter font for taclets (as in `ifElseSplit`) and sans-serif font for rules not given as taclets (as in ifElseSplit) to distinguish those concepts. In the remainder of this thesis, we will not make this distinction and always refer to rules (regardless of whether they are implemented as taclets of not) using the latter convention (sans-serif font). ◊

2.2.8 The Java Modeling Language

The Java Modeling Language (JML) [Lea+13; Ahr+16] is a specification language for Java used to describe the behavior of Java classes and methods. JML specifications are embedded into Java code via comment lines starting with an "@" sign. JML specifications serve as *contracts* for their implementers, defining what they can rely upon and what they have to deliver. The most prominent contract type in JML are *method contracts*; additionally, it is possible to define a *block contract* for an individual block below method level.[3] A contract consists of several *specification cases* connected by "**also**". Expectations, in other words *preconditions*, are specified using the **requires** keyword in a specification case; for postconditions, one uses **ensures**. *Exceptional behavior* is specified using the **signals** and (optional) **signals_only** keywords.

We explain these concepts along an example derived from [Ahr+16]. The method in Listing 2.3 (Page 37) computes the average of its argument, an array of integers. If this array is empty, then in Line 21, an `ArithmeticException` is thrown due to a division by 0. The JML contract of method `avg` contains two specification cases. The first defines that for a non-empty method parameter, the result, addressed by the expression **\result**, has to equal the average, which is specified using the summation operator **\sum** (one of several (generalized) quantifiers in JML, apart from, e.g., **\forall**, **\exists**, **\min**). The other ("**also**") case applies for empty parameter arrays, and states that then, an exception has to be thrown ("**exceptional_behavior**") which has to be a subtype of `ArithmeticException` ("**signals_only**"), and we give no guarantees about the resulting state (expressed by the **true** in the "**signals**" clause).

To reason about loops with symbolic guards, JML supports specifying *loop invariants*. A loop invariant is formula which holds upon each (re-)entry of the loop, and can thus be used to abstract away from the loop. *Inductive* loop invariants are strong enough to prove a method's postcondition. Sect. 2.3 expands on this intuition and shows how loop invariants are used in KeY proofs. In JML, one annotates a loop with an invariant using the **loop_invariant** keyword. To reason about total correctness, an additional

[3] Our specification language for AE builds on block contracts.

```
taclet ::=
  identifier "{"
    localSchemaVarDecl *
    assumptions ?
    findPattern ?
    applicationRestriction ?
    variableConditions ?
    ( goalTemplateList | "\closegoal" )
    ruleSetMemberships ?
  "}"

localSchemaVarDecl ::= "\schemaVar" schemaVarDecl
schemaVarDecl ::= schemaVarType identifier ( "," identifier ) * ";"

findPattern ::= "\find" "(" schematicExpression ")"
schematicExpression ::= schematicSequent | schematicFormula | schematicTerm

applicationRestriction ::= "\inSequentState" | "\sameUpdateLevel"
     | "\antecedentPolarity" | "\succedentPolarity"
     | "\notInAbstractUpdateScope"

variableConditions ::= "\varcond" "(" variableConditionList ")"
variableConditionList ::= variableCondition ( "," variableCondition ) *
variableCondition ::= "\new" "(" identifier ","
       ( "\dependingOn" "(" identifier ")" | type ) ")"
     | "\notFreeIn" "(" identifier "," identifier ")" | ...

goalTemplateList ::= goalTemplate ( ";" goalTemplate ) *
goalTemplate ::=
    branchName ?
    ( "\replacewith" "(" schematicExpression ")" ) ?
    ( "\add" "(" schematicSequent ")" ) ?
    ( "\addrules" "(" taclet ( "," taclet ) * ")" ) ?
branchName ::= string ":"

ruleSetMemberships ::= "\heuristics" "(" identifierList ")"
identifierList ::= identifier ( "," identifier ) *
```

Figure 2.7: Taclet Syntax (Source: [Ahr+16])

Listing 2.3: JML Example: Class Average

```
class Average {
  /*@ public normal_behavior
    @ requires nums.length > 0;
    @ ensures \result ==
    @     (\sum int i; 0 <= i && i < nums.length; nums[i])
    @   / nums.length;
    @ also
    @ public exceptional_behavior
    @ requires nums.length == 0;
    @ signals_only ArithmeticException;
    @ signals (ArithmeticException) true;
    @*/
  public int avg(int[] nums) {
    int sum = 0, i = 0;

    while (i < nums.length) {
      sum += nums[i];
      i++;
    }

    return sum / nums.length;
  }
}
```

decreases annotation has to be supplied, (usually) containing an integer term which strictly decreases in every iteration, without ever getting negative. Additionally, we specify with an **assignable** annotation the *loop frame*, i.e., which parts of the heap may be changed by the loop. A complete specification of the loop in method avg is given in Listing 2.4. It is strong enough to prove the two specification cases of avg.

Listing 2.4: Loop Specification for Class Average

```
/*@ loop_invariant i >= 0 && i <= nums.length &&
  @   sum == (\sum int k; 0 <= k && k < i; nums[k]);
  @ decreases nums.length - i;
  @ assignable nums[*];
  @*/
while (i < nums.length) {
  sum += nums[i];
  i++;
}
```

Block contracts [Wac12] are a KeY-specific concept replacing JML's **assert** and **assume** statements.[4] In principle, the behavior of any Java block can be specified in the same way as a method is specified [Ahr+16]. Block contracts have been extended in [Lan18] by additional behaviors, s.t. one can specify, e.g., when the content of a block completes due to a **break** statement, and which guarantees are provided in that case. In our implementation of AE, we heavily use these extensions.

If information needed in specifications is not completely provided by the source code itself, specification-only variables called *ghost variables* can be added. Those are declared in JML comments using the syntax "**ghost** *type name* = *value*;", where the initial assignment is optional. For example, one may write "**ghost \bigint** cost = 42;", where **\bigint** is a "specification-only" type representing the *mathematical* integers that cannot be used in normal variable declarations. The value of a ghost variable can be updated using the keyword **set**, as in "**set** cost = 0;".

We only presented essential aspects of JML needed to keep this thesis self-contained. A more general, tool-independent introduction to JML is given in [Ahr+16, Chapter 7].

[4] KeY also supports **assert** statements; they are *translated to block contracts* (as postconditions of a contract for an empty block). Similarly, we added support for **assume** statements (as "free" postconditions).

2.3 The Loop Scope Method

In sequential program verification, and thus also in SE, loop treatment is a crucial issue. A simple way to process loops is *unwinding*: Given a statement **`while`** (e) *body*, pull out one iteration by transforming it to the (equivalent) statement[5]

$$\mathtt{if}\ (e)\ \{\ body\ \mathtt{while}\ (e)\ body\ \}$$

as performed by the following SE rule:

$$\text{loopUnwind}\ \dfrac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{if}\ (e)\ \{\ body\ \mathtt{while}\ (e)\ body\ \}\ \omega], \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{while}\ (e)\ body\ \omega], \Delta}$$

However, if the loop guard contains symbolic variables, there is most likely no *concrete* upper bound on the necessary number of iterations. Therefore, it is in general not possible to symbolically execute a program with loops *exhaustively* by unwinding.

In these situations, one can use techniques utilizing *loop invariants* for abstracting away from the concrete effects of a loop. SE rules for loops based on loop invariants are based on an inductive argument: If we can prove that

(1) a formula holds before the first iteration and

(2) given that it held before the n-th iteration, we can prove that it holds before the $n+1$-th iteration,

we can replace the loop by that formula for the remaining execution. A formula satisfying (1) and (2) is called *loop invariant*; a loop invariant which is of sufficient precision s.t. one can prove an otherwise correct postcondition is called *inductive* loop invariant. In classic Hoare logic [Hoa69], the iteration rule is used for loop invariant reasoning. When applied together with the rule of consequence ("cons.") and the composition rule, we can create a derived rule containing the mentioned proof cases "initially valid" (case (1)), "preserved" (case (2)) and "use case" (application of the invariant abstraction):

$$\text{cons.}\ \dfrac{\dfrac{\text{(initially valid)}}{P \rightarrow Inv} \quad \text{composition}\ \dfrac{\text{iteration}\ \dfrac{\dfrac{\text{(preserved)}}{\{Inv \land B\}\ body\ \{Inv\}}}{\{Inv\}\ \mathtt{while}\ (B)\ body\ \{Inv \land \neg B\}} \quad \dfrac{\text{(use case)}}{\{Inv \land \neg B\}\ p\ \{Q\}}}{\{Inv\}\ \mathtt{while}\ (\mathtt{B})\ body\ ;\ p\ \{Q\}}}{\{P\}\ \mathtt{while}\ (B)\ body\ ;\ p\ \{Q\}}$$

When using the above rule, the loop invariant *Inv* also has to encode all the information

[5] We assume that *body* contains no labeled or unlabeled **`break`** or **`continue`** statements.

of the previous precondition P that might be necessary to prove the use case; if Inv' only describes the behavior of the loop, the formula $Inv := P \wedge Inv'$ has to be used as an invariant to not lose any information about P. The JavaDL version of this "classic" loop invariant rule therefore has a slightly different shape:

$$\textsf{simpleLoopInvariant}\quad \frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}((Inv \wedge se \doteq TRUE) \rightarrow [body]Inv), \Delta & \text{(preserved)}\\ \Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}((Inv \wedge se \doteq FALSE) \rightarrow [\pi\ \omega]\varphi), \Delta & \text{(use case)}\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \mathtt{\textbf{while}}\ \ (se)\ \ body\ \ \ \omega]\varphi, \Delta}$$

Here, the update $\mathcal{U}$ and context Γ, Δ is not removed, and in the "preserved" part only the actual loop invariant has to be proven. This is sound thanks to the introduction of *anonymizing updates* $\mathcal{U}_{havoc}$ erasing (both local variable and heap) locations assigned in the loop body. Thus, the "preserved" case still reasons about an *arbitrary* loop iteration, while at the same time, context information preserved by the loop is maintained.

The rule simpleLoopInvariant only works for "simple" expressions *se* without side effects, and normally completing loop bodies *body*. The reason for the latter restriction is that in the "preserved" case, *body* is executed outside the program context $\pi\ \omega$. Consequently, information about how to handle **`break`**, **`continue`** and **`return`** statements as well as thrown exceptions is no longer present. The approach originally implemented in the KeY system and described in [Ahr+16] builds on simpleLoopInvariant and uses heavy *program transformation* to address abrupt completion. It wraps the loop body in a labeled **`try`**-**`catch`** statement; **`break`**s, **`return`**s and **`continue`**s are transformed into labeled **`break`**s targeting the **`try`**-**`catch`** statement before which corresponding flags are set that describe the respective nature of the loop completion. Thrown exceptions are caught in the catch block and assigned to a new variable which makes exception objects available in the postcondition of the *preserved* branch. The loop guard is executed in total *four times* to capture side effects at all relevant places. The "preserved" case is not always what it seems: The invariant must be proven in the "preserved" case in cases where the loop body completes normally or abruptly continues; *the postcondition* must be shown in the "preserved" case when guard or body *complete abruptly for any other reason*.

The transformation-based approach has several downsides. It is hard to implement correctly since program transformation of Java code is generally intricate and error-prone. Indeed, new special cases uncovering so far undiscovered bugs in the KeY implementation continued being discovered, even though the rule has been in active use already for many years. Furthermore, proofs created with the rule are hard to understand (since due the program transformation of loop bodies, the connection to the original program gets blurry) and rather inefficient (since loop guards are executed multiple times and the separation into "preserved" and "use case" is not very meaningful in presence of abruptly completing loop bodies). Finally, the transformation-based loop invariant rule in KeY was never documented properly, since first, it is very difficult to write down in text-book style due to the program transformation, and secondly, it kept evolving due to necessary soundness or completeness fixes.

In [Was16], the concept of *loop scopes* was first mentioned, originally in the context of automatic loop invariant generation. Our work [SW17] describes the *loop scope invariant rule* for **while** loops and empirically shows that this rule leads to significant performance improvements in terms of proof sizes compared to the transformation-based rule. In this section, we describe syntax, semantics and calculus rules related to the loop scope statement; in Sect. 2.3.3, we discuss the performance of the loop scope invariant rule as well as a new implementation as a KeY *taclet* (as opposed to a built-in rule).

2.3.1 Syntax and Semantics of the Loop Scope Statement

Loop scope statements are a syntactic extension of Java and may legally appear (cf. Sect. 2.2.1) in JavaDL modalities. We define loop scope statements as follows:

Definition 2.13 (Loop Scope Statement)**.** Let `x` be a program variable of type `boolean` and p be a Java statement. A *loop scope statement* is a Java statement of the form **`loop-scope`** `(x) {`p`}`. We call `x` the *index* of the loop scope statement and p the *body* of the loop scope statement. $\Diamond$

In [Was16; SW17], an alternative notation for loop scope statements was used: $\circlearrowright_{\mathtt{x}}\ p\ {}_{\mathtt{x}}\circlearrowleft$. We use the more familiar ASCII variant for this thesis to emphasize that loop scopes are a Java extension and not part of the logical syntax. Intuitively, loop scope statements **`loop-scope`** `(x) {` p `}` arise from the symbolic execution of loop statements **`while`** `(`b`) {` p `}`. The index `x` encodes *completion information* about the loop: If it is **`false`** after execution, then the loop would continue with another iteration, and if it is **`true`**, it would terminate (either normally or abruptly). In contrast to [Was16; SW17], we do not allow the loop scope body p to complete normally; this is achieved by adding explicit **`continue`** and **`break`** keywords in the loop scope invariant rule as described in Sect. 2.3.2. According to this intuition, we define the semantics of loop scope statements following the style of the JLS [Gos+05].

Definition 2.14 (Semantics of Loop Scope Statements)**.** Let **`loop-scope`** `(x)` `{`p`}` be a loop scope statement for an index variable `x` and body p. It is executed by first executing the statement p. Then there is a choice:

- If execution of p completes normally, the behavior of the loop scope statement is undefined.
- If execution of p completes abruptly because of a **`break`** with no label, `x` is set to **`true`** and the loop scope statement completes normally.
- If execution of p completes abruptly because of a **`continue`** with no label, `x` is set to **`false`** and the program containing the loop scope statement terminates (not even executing surrounding **`finally`** blocks).
- If execution of p completes abruptly because of a **`continue`** with label l, then there is a choice:
  - If the loop scope statement has label l, then `x` is set to **`false`** and the program containing the loop scope statement terminates (not even executing surrounding **`finally`** blocks).
  - If the loop scope statement does not have label l, `x` is set to **`true`** and the loop scope statement completes abruptly because of a **`continue`** with label l.
- If execution of p completes abruptly for any other reason, `x` is set to **`true`** and the loop scope statement completes abruptly for the same reason. ◊

The case of abrupt completion because of a **`break`** with a label is handled by the general rule for labeled statements, similar to the definition of the semantics of a **`while`** statement in the JLS: "`l:`p" completes normally if p completes normally or because of a **`break`** with label `l`; if p completes abruptly for any other reason R, then "`l:`p" also completes abruptly for reason R. The only possibility for a loop scope statement to complete *normally* is the case where the body completes because of an unlabeled **`break`**. For unlabeled **`continue`** statements and **`continue`** statements with a label matching the label of the loop scope statement, the whole symbolic execution process is halted. In all other cases, the loop scope statement completes abruptly for the same reason as the completion of its body. We illustrate the semantics of loop scope statements along some examples.

Example 2.3 (Loop Scope Semantics)**.** Consider the program in Listing 2.5. It is semantically equivalent to the program "`y = 2; x = `**`false`**`;`" since the body of the loop scope completes abruptly because of a **`continue`** with label `l` after setting `y` to 2; because the loop scope statement has label `l`, the index variable `x` is set to **`false`** and the program

Listing 2.5: Example 1

```
try {
  l: loop-scope (x) {
    y = 2;
    continue l;
    f();
  }
} finally {
  y = 0;
}
```

Listing 2.6: Example 2

```
try {
  l: loop-scope (x) {
    y = 2;
    break l;
    f();
  }
} finally {
  y = 0;
}
```

Listing 2.7: Example 3

```
try {
  l: loop-scope (x) {
    y = 2;
    return y / 0;
    f();
  }
} finally {
  y = 0;
}
```

is exited. The **`finally`** block is not executed, thus the final value of `y` is 2 and *not* 0. In Listing 2.6, the body of the loop scope statement completes abruptly because of a **`break`** with label `l` after setting `y` to 2. Therefore, `x` is set to **`true`** and the loop scope statement completes because of a **`break`** with label `l`. Since the statement has label `l`, the whole *labeled* statement completes normally due to general rule for labeled statements. Afterward, the **`finally`** block is executed. Thus, the whole program is equivalent to "`y = 2; x = true; y = 0;`". The loop scope body in Listing 2.7 completes abruptly because of a **`throw`** of an object *exc* of type `ArithmeticException` after setting `y` to 2. It does *not* complete because of a **`return`**: The exception is raised while evaluating the returned value, and before the actual execution of the **`return`** instruction. Since the **`try`** statement does not contain a matching **`catch`** clause, the **`finally`** block is executed; since it completes normally, the whole **`try`**-**`finally`** block completes abruptly because of a **`throw`** of *exc*. The whole program is therefore roughly[6] equivalent to

```
y = 2; x = true; y = 0;
throw new ArithmeticException();
```

The program "`loop-scope(x) {}`" is not well-defined since the loop scope body completes normally. The same holds, e.g., for "`loop-scope(x) { l: break l; }`". ◊

In the subsequent section, we devise JavaDL SE calculus rules reflecting the semantics of loop scope statements as in Def. 2.14. Afterward, we introduce the loop invariant rule for **`while`** loops based on loop scopes and argue for its soundness.

[6] With the obvious difference in the exception message, etc.

2.3.2 Loop Scope SE Rules and the Loop Scope Invariant Rule

To symbolically execute loop scope statements, we need to devise SE rules for all the abrupt completion cases in Def. 2.14. Loop scope openings "**`loop-scope`** `(x) {`" are part of the non-active prefix π in SE rules, corresponding closing braces are, as usual, part of the postfix ω. Figure 2.8 shows the *complete* set of loop scope SE rules; there is, in particular, no rule for a loop scope statement with empty body. It is straightforward to see that these rules faithfully model the semantics described in Def. 2.14. In the two cases where the loop scope index is set to **`false`** and the program terminates entirely, lsContinue and lsLblContinueMatch, we directly introduce an update application "{`x` := FALSE}" instead of the equivalent modality "[`x =` **`false`**`;`]" to make clear that the program terminated. The rule for labeled breaks, lsLblBreak, puts the newly generated program inside a new block (wraps it in curly braces) to achieve that the result is a well-formed program if the loop scope statement itself had a label. Compared to [SW17], we removed the rule for the empty loop scope (reflecting our likewise updated loop scope semantics) and added one for labeled **`continue`**s with a non-matching label (or no label) in front of the loop scope statement. The latter was not necessary before since the loop scope-based invariant rule in [SW17] pushed loop labels inside loop scopes (which we do not do), and thus only labeled **`continue`**s with non-matching labels could become active statements within a loop scope statement due to prior transformations by other SE rules.

The loop invariant rule based on loop scopes, loopScopeInvariant, is shown in Fig. 2.9.[7] The "initially valid" case equals the corresponding one of simpleLoopInvariant. Anonymizing updates are employed accordingly. The most striking difference is that the rule no longer has three premises, but *only two*: The "preserved" and "use case" premises are combined to one premise "preserved & use case". This has the advantage that loop guards only have to be executed *once* (instead of four times in the transformation-based variant) and confusing cases where in the preserved case, one has to prove the postcondition instead of the invariant since the loop body completed abruptly, do not occur.

The main advantage of the rule, though, is that the loop body *body* is *not transformed at all*. Instead, it is wrapped in a loop scope statement with a fresh index variable `x` and an **`if`** statement which has the loop guard *nse* as guard. After *body*, a trailing **`continue`** statement is inserted to signal that at this place the loop would continue with another iteration; for the case that the loop guard does not hold (i.e., the loop would normally terminate at this point), we add a **`break`** statement to the **`else`** branch of the conditional. As opposed to [SW17], we do not need to match an optional leading loop label, which is

[7] Note that, for all invariant rules using anonymizing updates, the invariant formula *Inv* has to contain a conjunct asserting that the update $\mathcal{U}_{havoc}$ anonymizes at least the part of the heap that is assigned.

$$\text{lsBreak}\ \frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ \textbf{true};}\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{loop-scope}\ (x)\ \{\ \textbf{break};}\ p\ \mathtt{\}}\ \omega]\varphi, \Delta}$$

$$\text{lsContinue}\ \frac{\Gamma \vdash \{\mathcal{U}\}\{\mathtt{x} := \mathrm{FALSE}\}\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{loop-scope}\ (x)\ \{\ \textbf{continue};}\ p\ \mathtt{\}}\ \omega]\varphi, \Delta}$$

lsLblContinueMatch
$$\frac{\Gamma \vdash \{\mathcal{U}\}\{\mathtt{x} := \mathrm{FALSE}\}\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ l\mathtt{:\ \textbf{loop-scope}\ (x)\ \{\ \textbf{continue}}\ l\mathtt{;}\ p\ \mathtt{\}}\ \omega]\varphi, \Delta}$$

lsLblContinueNoMatch
$$\frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ \textbf{true};\ \textbf{continue}}\ l\mathtt{;}\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ (l'\mathtt{:})^?\ \mathtt{\textbf{loop-scope}\ (x)\ \{\ \textbf{continue}}\ l\mathtt{;}\ p\ \mathtt{\}}\ \omega]\varphi, \Delta}\ l \neq l'$$

$$\text{lsLblBreak}\ \frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\{\ x\ =\ \textbf{true};\ \textbf{break}}\ l\mathtt{;\ \}}\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{loop-scope}\ (x)\ \{\ \textbf{break}}\ l\mathtt{;}\ p\ \mathtt{\}}\ \omega], \Delta\varphi}$$

$$\text{lsThrow}\ \frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ \textbf{true};\ \textbf{throw}}\ se\mathtt{;}\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{loop-scope}\ (x)\ \{\ \textbf{throw}}\ se\mathtt{;}\ p\ \mathtt{\}}\ \omega], \Delta\varphi}$$

$$\text{lsReturnNonVoid}\ \frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ \textbf{true};\ \textbf{return}}\ se\mathtt{;}\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{loop-scope}\ (x)\ \{\ \textbf{return}}\ se\mathtt{;}\ p\ \mathtt{\}}\ \omega], \Delta\varphi}$$

$$\text{lsReturnVoid}\ \frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ \textbf{true};\ \textbf{return};}\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{loop-scope}\ (x)\ \{\ \textbf{return};}\ p\ \mathtt{\}}\ \omega], \Delta\varphi}$$

Figure 2.8: Calculus Rules for Loop Scope Statements

$$
\text{loopScopeInvariant}\quad
\frac{\begin{array}{ll}
\Gamma \vdash \{\mathcal{U}\}\mathit{Inv}, \Delta & \text{(initially valid)} \\
\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}\big(\mathit{Inv} \rightarrow [\pi & \text{(preserved \& use case)} \\
\quad \texttt{loop-scope (x) \{} & \\
\quad\quad \texttt{if (}\mathit{nse}\texttt{) \{} & \\
\quad\quad\quad \mathit{body} & \\
\quad\quad\quad \texttt{continue;} & \\
\quad\quad \texttt{\} else \{} & \\
\quad\quad\quad \texttt{break;} & \\
\quad\quad \texttt{\}} & \\
\quad \texttt{\}}\ \ \omega]((\mathtt{x} \doteq \mathit{TRUE} \rightarrow \varphi) \land (\mathtt{x} \doteq \mathit{FALSE} \rightarrow \mathit{Inv}))\big), \Delta &
\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{while (}\mathit{nse}\texttt{)}\ \mathit{body}\ \ \omega]\varphi, \Delta}\ \mathtt{x}\ \text{fresh}
$$

Figure 2.9: The Loop Scope Invariant Rule for Java

matched by rule lsLblContinueMatch when we reach a corresponding labeled **`continue`**. In [SW17], the loop label had to be pushed to *inside* the loop scope.

The loop scope index x is then used in the new postcondition $(\mathtt{x} \doteq \mathit{TRUE} \rightarrow \varphi) \land (\mathtt{x} \doteq \mathit{FALSE} \rightarrow \mathit{Inv})$ to distinguish between the cases where the loop was left (and therefore x is TRUE) and the original postcondition φ has to be proven, and the cases where the loop would continue with another iteration (and therefore x is FALSE) and the loop invariant formula *Inv* has to be proven. This explains why the two rules for unlabeled **`continue`** and labeled **`continue`** with matching label in Fig. 2.8 are designed to terminate the surrounding program: In loop invariant-based reasoning, one has to prove the loop invariant for the effects of the loop body in isolation. If we did not exit the program, program variables in *Inv* could be changed by the remaining program in ω, rendering the rule unsound. We point out that in loopScopeInvariant, there are no hidden program transformations nor restrictions like the absence of side effects or abrupt completion in the loop guard. The rule is easy and straightforward to document in a text book-style format, and much less brittle in comparison to the transformation-based rule. In Sect. 2.3.3, we also show that this rule is easier to implement as a KeY *taclet* which is better readable and maintainable than a hard-coded Java implementation.

The following theorem from [SW17] states the validity of the loopScopeInvariant rule.

Theorem 2.3 (Soundness of loopScopeInvariant)**.** *The rule* loopScopeInvariant *is* sound*,* *i.e., if the "initially valid" and "preserved & use case" premises are valid, then also the conclusion is valid.*

We omit a proof of Thm. 2.3 since it is not in the scope of this thesis. A proof sketch is contained in [SW17]. The basic insight, apart from the usual inductive argument for loop invariant rules, is that the semantics of loop scopes and the added **`continue`** and **`break`** statements ensure that always the correct proof obligations are chosen, i.e., either the invariant is proven when the loop would continue with another iteration, or SE continues and finally the postcondition is proven when the loop would have been exited.

So far, we only showed the rule for the box modality (partial correctness). It can be extended to total correctness by adding a termination witness, a so-called *loop variant*. A variant is usually an integer term which strictly decreases in each loop iteration and never gets negative. Figure 2.10 shows this extension ("loopScopeInvariantDia"). The initial value of the variant term *variantTerm* is stored in variable `variant`; before each further iteration, the additional proof obligation *variantTerm* $\prec$ `variant` has to be shown. The predicate symbol $\prec : \top \times \top$ used in such termination proofs is axiomatized as a well-founded relation. KeY does not only support integer variants, but also has built-in axioms for lexicographic ordering of pairs and finite sequences [Ahr+16, Sect. 9]. We illustrate the loop scope invariant rule in the diamond version along a concrete example.

loopScopeInvariantDia

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}\{\mathtt{variant} := variantTerm\}\big(Inv \rightarrow \langle\pi & \text{(preserved \& use case)}\\ \quad \mathbf{loop\text{-}scope}\ (\mathtt{x})\ \{ & \\ \quad\quad \mathbf{if}\ (nse)\ \{ & \\ \quad\quad\quad body & \\ \quad\quad\quad \mathbf{continue}; & \\ \quad\quad \}\ \mathbf{else}\ \{ & \\ \quad\quad\quad \mathbf{break}; & \\ \quad\quad \} & \\ \quad \}\ \ \omega\rangle((\mathtt{x} \doteq TRUE \rightarrow \varphi) \land & \\ \quad\quad\quad (\mathtt{x} \doteq FALSE \rightarrow (Inv \land variantTerm \prec \mathtt{variant})))\big), \Delta & \end{array}}{\Gamma \vdash \{\mathcal{U}\}\langle\pi\ \mathbf{while}\ (nse)\ body\ \ \omega\rangle\varphi, \Delta}\ \mathtt{x}\text{ fresh}$$

Figure 2.10: The Loop Scope Invariant Rule with Termination for Java

Example 2.4 (Loop Scope Invariant Rule)**.** We consider a linear search method as displayed in Listing 2.8 as an example. The method accepts an array `arr` of **`int`** values

Listing 2.8: Linear Search

```
/*@ public normal_behavior
  @ ensures ((\exists int i; i >= 0 && i < arr.length; arr[i] == elem) ==>
  @             arr[\result] == elem &&
  @             (\forall int j; j >= 0 && j < \result; arr[j] != elem)) &&
  @         ((\forall int i; i >= 0 && i < arr.length; arr[i] != elem) ==>
  @             \result == -1);
  @*/
public int linearSearch(int[] arr, int elem) {
  int i = 0;
  /*@ loop_invariant
    @     i >= 0 && i <= arr.length
    @   && (\forall int j; j >= 0 && j < i; arr[j] != elem);
    @ decreases arr.length - i;
    @ assignable \nothing;
    @*/
  while (i < arr.length) {
    if (arr[i] == elem) {
      return i;
    }

    i++;
  }

  return -1;
}
```

$$
\begin{array}{l}
\Gamma \vdash \{\texttt{\_arr} := \texttt{arr} \,\|\, \texttt{\_elem} := \texttt{elem} \,\|\, \texttt{exc} := \texttt{null} \,\|\, \texttt{i} := 0\} \\
\quad \{\texttt{i} := i_0\} \\
\quad\; \{\texttt{variant} := \texttt{\_arr.length} - \texttt{i}\}\big(\\
\qquad \texttt{i} \geq 0 \land \texttt{i} \leq \texttt{\_arr.length} \land \forall j;\ (j \geq 0 \land j < \texttt{i} \land \texttt{\_arr}[j] \not\doteq \texttt{\_elem}) \rightarrow
\end{array}
$$

```
⟨try { method-frame(result->result,
                    source=linearSearch(int[],int)@LinearSearch,
                    this=self) : {
    loop-scope (x) {
      if (i < _arr.length) {
        {
          if (_arr[i] == _elem) {
            return i;
          }
          i++;
        }
        continue;
      } else {
        break;
      }
    }
    return -1;
}} catch (java.lang.Throwable e) {
  exc = e;
})
```

$$
\begin{array}{l}
\rangle((\texttt{x} \doteq \mathit{TRUE} \rightarrow \mathit{Post} \land \texttt{exc} \doteq \texttt{null}) \land \\
\quad (\texttt{x} \doteq \mathit{FALSE} \rightarrow \\
\qquad (\texttt{i} \geq 0 \land \texttt{i} \leq \texttt{\_arr.length} \land \forall j;\ (j \geq 0 \land j < \texttt{i} \land \texttt{\_arr}[j] \not\doteq \texttt{\_elem}) \land \\
\qquad \texttt{\_arr.length} - \texttt{i} \prec \texttt{variant})))\big), \Delta
\end{array}
$$

Figure 2.11: Example Application of the Loop Scope Invariant Rule

and another **int** value elem as parameters. Its contract (Lines 2 to 6) specifies that if, and only if, elem is contained in arr, the smallest index of elem in arr is returned; otherwise, the returned result is -1. The implementation is straightforward: A loop linearly inspects the array elements; if elem is found, the current index is directly returned. For proving the method correct, a loop invariant is needed, since the length of the array is unknown (and therefore, unwinding is not an option). Since the invariant only has to hold *before another loop iteration*, it only specifies, apart from basic constraints on the bounds of index i (Line 11), that elem has *not yet been found* (Line 12). If during an iteration, the searched element is found, it is directly returned. In this case, the loop therefore does not continue with another iteration. Consequently, the invariant needs not hold. The loop specification also defines a loop variant needed for proving loop termination (the **decreases** term in Line 13) and an **assignable** location set in Line 14. KeY automatically extracts assignable program variables, so the **assignable** specification only needs to list assignable *heap locations*; since the loop does not change the heap, **\nothing** is an adequate specifier at this place.

When using the transformation-based loop invariant rule, proving the "preserved" branch requires showing two distinct cases: One where i is incremented because elem has not been found and the invariant has to be proven, and one where the **return** statement has been executed and the *postcondition* has to be shown. The latter contradicts the intuition of the "preserved" branch, since it is unrelated to preservation.

An application of the loopScopeInvariant rule on the **while** loop in Line 16 leads to *two* branches. In the first branch, it has to be shown that the loop invariant holds for the initial state where i is 0. The initial sequent for the second branch comprising the "preserved" and "use case" cases is shown in Fig. 2.11. We omitted the framing formula generated for the **assignable** specification and applied some further simplifications to increase readability. In particular, we abbreviate the postcondition with *Post* and concrete premises with Γ, Δ. Apart from that, the sequent for the most part looks like the one generated by the KeY implementation. The variable i is anonymized by a fresh constant i_0; concrete loop invariant and variant terms are shown. The program inside the modality contains a method frame which is in particular needed to interpret **return** statements in this example, and a surrounding **try** statement to handle uncaught exceptions.

In the branch starting from the described sequent, three interesting subbranches (which cannot trivially be closed by application of the preconditions, like exceptions for wrong array accesses) arise: (1) execution of the **return** statement; here, the loop scope index x is set to TRUE and the postcondition has to be shown, (2) execution of the **continue** statement; here, the loop scope index x is set to FALSE, the program is left and the loop invariant and variant have to be shown, (3) execution of the **break** statement; here, the loop scope index x is set to TRUE and the postcondition has to be shown. The corresponding SE rules for these cases are IsReturnNonVoid, IsContinue and IsBreak. We point out that already for this simple example, a proof with the loop scope invariant rule (1,067 proof nodes, 53 SE steps) is significantly shorter than a corresponding one with the transformation-based rule (1,337 proof nodes, 89 SE steps): We observe a reduction of 40,5 % in the number of SE steps. Sect. 2.3.3 provides a more intensive and systematic analysis of the performance of the loop scope invariant rule. ◊

Finally, we want to briefly address the negative implications of the loop scope invariant rules having only two (and not three) branches. While this is beneficial in terms of efficiency (apart from the fact that we do not need program transformation, which enabled us to create sound and understandable loop invariant rules for Java), it is sometimes helpful to have a clear separation between the *preserved* and *use case* branches, especially in *teaching*. In practice, one can always assign a proof branch to one of those cases by inspecting the value of the loop scope index variable. For usage in a formal methods course, the separation into three branches might nevertheless be useful. We propose a straightforward variant of the loop scope invariant rules to three-branch versions; the rule for the box modality, threeBranchLoopScopeInvariant, is shown in Fig. 2.12. The idea is simply to divide the postcondition into two parts, one treating the case where the loop scope index x is FALSE and one where it is TRUE, and to assign to each case a separate proof branch. The remaining sequent stays the same. This rule is sound, but not efficient: The body and loop guard are executed twice. We point out that there is no easy solution clearly separating these cases while sparing the associated redundancy due to the presence of abrupt completion. One partial solution could be to not use loop *invariants*, but full *loop contracts* [Tue10; Lan18] as specifications for loops (see Sect. 6.2 for a discussion of the related "super invariants"). Those contracts could also specify under which conditions abnormal completion does occur, allowing to close irrelevant proof/SE branches early. For use in teaching, threeBranchLoopScopeInvariant is still interesting since regarded problems are usually small and efficiency is no big issue. In bigger case studies, we recommend using the two-branch rules.

2.3.3 Performance of the Loop Scope Technique

In [SW17], we evaluated our implementation of the loop scope-based loop invariant rule by proving a set of benchmark problems both with the previous program transformation-based invariant rule and the new one. As benchmarks, we chose all examples shipped threeBranchLoopScopeInvariant

$$
\frac{
\begin{array}{ll}
\Gamma \vdash \{\mathcal{U}\}Inv, \Delta & \text{(initially valid)} \\
\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}\big(Inv \rightarrow [\pi & \text{(invariant preserved)} \\
\quad \texttt{loop-scope (x) \{} & \\
\quad\quad \texttt{if } (nse) \texttt{ \{} & \\
\quad\quad\quad body & \\
\quad\quad\quad \texttt{continue;} & \\
\quad\quad \texttt{\} else \{} & \\
\quad\quad\quad \texttt{break;} & \\
\quad\quad \texttt{\}} & \\
\quad \texttt{\}} \;\; \omega](\mathtt{x} \doteq FALSE \rightarrow Inv))\big), \Delta & \\
\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}\big(Inv \rightarrow [\pi & \text{(use case)} \\
\quad \texttt{loop-scope (x) \{} & \\
\quad\quad \texttt{if } (nse) \texttt{ \{} & \\
\quad\quad\quad body & \\
\quad\quad\quad \texttt{continue;} & \\
\quad\quad \texttt{\} else \{} & \\
\quad\quad\quad \texttt{break;} & \\
\quad\quad \texttt{\}} & \\
\quad \texttt{\}} \;\; \omega](\mathtt{x} \doteq TRUE \rightarrow \varphi))\big), \Delta &
\end{array}
}{
\Gamma \vdash \{\mathcal{U}\}[\pi \;\; \texttt{while} \;\; (nse) \;\; body \;\; \omega]\varphi, \Delta
} \; \mathtt{x} \text{ fresh}
$$

Figure 2.12: The Three-Branch Loop Scope Invariant Rule (Box Version)

with the KeY system containing loops, which range between 27 and 640 SE steps (using the old invariant rule); in average, it takes 170 SE steps to execute one example. The evaluation showed that the loop scope approach helps to significantly reduce proof sizes, and, to an even greater extent, the numbers of SE steps.

Since then, we replaced the old implementation of the loop scope invariant rule, which was programmed purely in Java, by a new version written as a KeY *taclet*. Moreover, we adapted the implementation to reflect the changes to the semantics of loop scopes as presented in this chapter (e.g., the semantics of loop scopes is now undefined for normally completing bodies, which is different to [SW17]). To assess the effects of these changes and of other developments in the KeY system during the past three years, we repeated the evaluation in our new system for the same sample set.[8]

As a baseline, we again chose the transformation-based rule. In addition, we also describe the effects of only considering larger examples with more than 100 SE steps (which amounts to removing 20 of 51 examples), and show the results for evaluations *with and without one-step simplification*. The latter is an aggregator rule accumulating taclet applications of specific rule sets for low-level simplification of sequents. [Ahr+16, Sect. 4.3.1]. The original experiment was conducted without using this rule. Since the one-step simplifier can occasionally change the proof structure in non-trivial ways, we were interested in the effects of using it in our setup.

Figure 2.13 contains box plots for the percentage difference of the numbers of proof nodes and SE steps between the transformation- and loop scope-based rules without, and Fig. 2.14 with one-step simplification. The bars in the middle of the boxes represent the median, the boxes itself the midspread (the middle 50%), and the whiskers point to the last items that are still within 1.5 of the inter quartile range of the lower/upper quartile. The examples which are not covered by the whiskers, the outliers, are signified as points. Exact numbers are in Tables A.1 and A.2 in Appendix A.

Results Without One-Step Simplification The results without one-step simplification are similar to those reported in [SW17]. We saved between 5.5% and 70% of SE steps (3% and 63% in [SW17]); for 50% of the examples, we could reduce the number of SE steps by 15% to 30% (17% and 32% in [SW17]). Figure 2.13c shows the corresponding box plot. Excluding problems with less than 100 SE steps (Fig. 2.13d) slightly reduces these numbers and removes one outlier; the overall picture stays the same. The savings

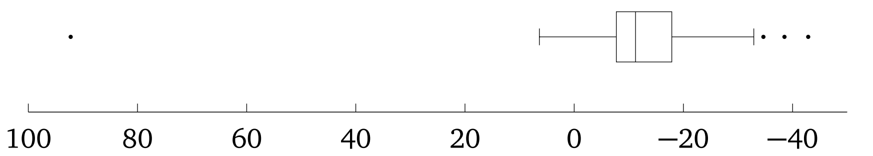
[8] We removed three examples which were erroneously added to the benchmark set in [SW17]. Two of them in fact did not include loops, and one could be closed without applying SE steps. Excluding these examples did not noticeably change the results of our evaluation.

are mostly due to the overhead of executing loop guards four times in the transformation-based rule. While we *save SE steps for all examples* when using the loop scope invariant rule, there are five problems where *proof sizes increase* (box plot in Fig. 2.13a). In four of them, proofs grow mildly between 2.3% and 6.4%. However, there is one negative outlier ("coincidence count") whose proof *grows by 92.2%,* although we saved 28% of SE steps. This is less than reported in [SW17]. There, the proof of the same outlier, where using the loop scope-based rule also had the worst effect, grew by 259%.

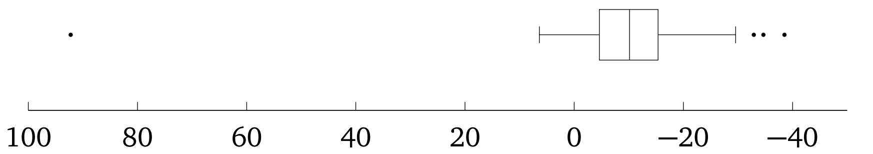
The reasons are still the same: KeY's strategies make better decisions in comparable proof situations (sequents have the same formulas, but with different order and age) after an application of the old invariant rule. There are some interesting positive outliers: The "lcp" example has an extremely complicated loop guard, which is why the loop scope invariant rule excels here. Example "ArrayList.remove.0" contains two nested loops. It is not necessary to apply an invariant rule to the inner loop to close the proof; still, KeY chooses an invariant rule, since it has low cost. This choice leads to a lower degree of branching, and in total to a smaller proof, in case of the new rule.

Not considering small proofs with less than 100 SE steps has almost no effect (Fig. 2.13b).

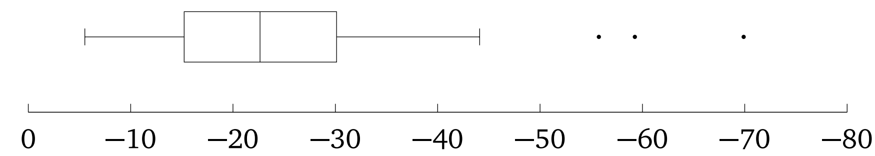
Results with One-Step Simplification We complemented the evaluation of [SW17] by an additional set of runs with activated one-step simplifier. The results emphasize the generally positive effect of using the loop scope-based invariant rule, and also demonstrate the extent to which one-step simplification can change proof sizes and structure. Figure 2.14 contains the box plots for this data, complete results are in Table A.2 in Appendix A. The numbers of saved SE steps decrease a little: The median percentage of saved SE steps is 20.75% with one-step simplification as opposed to 22.64% without. However, there is now only one example where the proof size increases when using the new loop invariant rule (the normal behavior specification case for the get method of a LinkedList implementation). The previously worst performing problem, coincidence count, now has a positive outcome, with a proof size *reduction* of 16.4% (as opposed to an increase of 92.2% without one-step simplification). Instead of three positive outliers, we have five, since in case of two problems (one of medium and one of smaller size in terms of SE steps), we managed to decrease proof sizes by over 50%. We manually inspected the proof for the one negative (in terms of proof size) example. In this example, the "preserved" cases are comparatively small; the loop guards are simple inequations without side effects, and loop bodies do not complete abruptly. The additional number of nodes accrues after SE in the use case. Here, KeY postpones some crucial simplifications to after splitting the proof by case distinctions when using the new rule; for the old rule, simplifications are applied



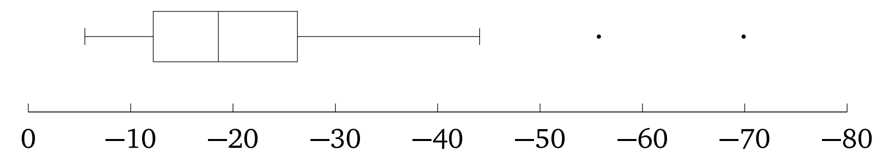
(a) Proof Nodes



(b) Proof Nodes (Over 100 SE Steps)



(c) Symbolic Execution Steps



(d) Symbolic Execution Steps (Over 100 SE Steps)

Figure 2.13: Percentage Difference in Number of Proof Nodes / SE Steps, without One-Step Simplifier

earlier, saving double rule applications. Restriction to problems with over 100 SE steps removes two of the positive outliers in the proof size statistics and one positive outlier in the SE steps statistics, and does not significantly affect the overall picture.

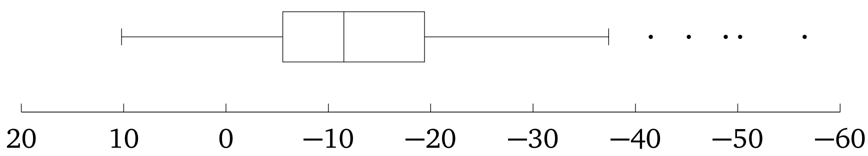
Taclet Implementation We implemented the new version of the loop scope invariant rule as a *taclet* in KeY. Listing A.1 in Appendix A shows the taclet code for total correctness. Quoting [Ahr+16], "loop invariant rules are probably the most involved and complex rules of the KeY system's JavaDL calculus".[9] We think that by implementing this complex rule as a taclet, we contributed to the understandability, maintainability and extensibility of KeY. In fact, we discovered an (uncritical and not soundness-relevant) bug in the implementation of the old rule, which became explicit only in the textual taclet representation. This is briefly discussed in Listing A.1. Moreover, adding the three-branch loop scope invariant rule (Fig. 2.12) took less than a minute, as we only needed to copy a taclet, change its name, duplicate the definition of the second child node, and delete the first and second half, respectively, of the postconditions for those nodes. When implementing the first invariant taclet, we added five new variable condition types and six term transformers. Although those are implemented as Java classes, responsibilities are clearer and better separated in comparison to a monolithic implementation of a Java built-in rule; moreover, those conditions and transformers can easily be used in other rules, e.g., variants or extensions of existing loop invariant rules.

2.4 Completion Scopes

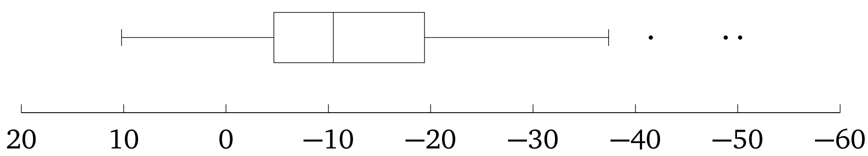
Java [Gos+05] is a complex sequential programming language in which statements do not always "just" complete: Rather than completing normally, they might complete *abruptly* for a variety of reasons. The best known reason for abrupt completion is because of an exception. Frequently, programmers need to react to the event of a thrown exception. In case of checked exceptions, this is even mandatory. The Java language offers a special statement type to this end: The **`try`**-**`catch`**-**`finally`** statement with the well-known semantics of first executing the **`try`** block, then, in the case of abrupt completion due to an exception, executing the **`catch`** block with matching exception type, and finally executing the **`finally`** block (whatever happened before).

A Java statement can complete abruptly for a number of reasons apart from a thrown exception. The complete list of reasons for abrupt completion of a statement is [Gos+05]:

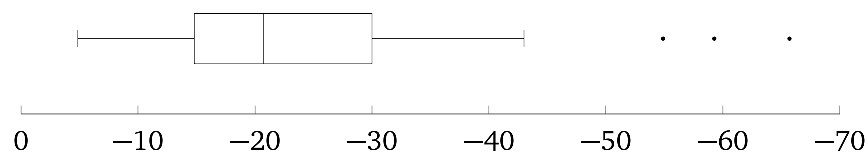
[9] Arguably, this does not hold anymore after the implementation of Abstract Execution in KeY.



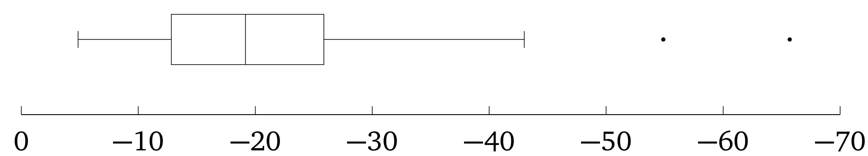
(a) Proof Nodes



(b) Proof Nodes (Over 100 SE Steps)



(c) Symbolic Execution Steps



(d) Symbolic Execution Steps (Over 100 SE Steps)

Figure 2.14: Percentage Difference in Number of Proof Nodes / SE Steps, with One-Step Simplifier

- a **`break`** with no label,
- a **`break`** with a given label,
- a **`continue`** with no label,
- a **`continue`** with a given label,
- a **`return`** with no value,
- a **`return`** with a given value,
- a **`throw`** with a given value, including exceptions thrown by the Java Virtual Machine.

The evaluation of expressions can also complete abruptly, but only due to an exception or error. Sometimes, and especially in program analysis, one needs to react also to other types of abrupt completion than exceptions. Method frames (Def. 2.2) can be seen as a way of reacting to **`return`** statements inside a method body (although they also serve other purposes). The loop scope statements introduced in Sect. 2.3 are basically a way of reacting to abrupt completion inside a loop body. However, method frames are too complex to be used for the sole purpose of "catching" returns, and the degree of information reported to the outside by loop scopes (whether or not the loop would continue with another iteration) is quite coarse. Attempts to use loop scopes for **`for`** loops [WS19; Dre19], where in the case of continued looping, the update sequence of the **`for`** loop has to be executed, can be considered as failed. The resulting invariant rule looks quite inelegant, due to the fact that loop scopes have not been developed for this kind of application. Furthermore, proofs based on the rule are less efficient than when using the alternative approach of first transforming **`for`** to **`while`** loops [Dre19]. Finally, some of the loop scope rules had to be adapted to fit into the new framework; those changes were later on discovered to be unsound if loop scope statements were not used as intended.[10] Another concern is the low "transparency" achieved by loop scopes. The semantics, especially the effect on the loop scope index and, for instance, the leaving of the whole modality after a **`continue`** statement, are hidden inside the definition of the eight loop scope rules depicted in Fig. 2.8. For this reason, it is difficult to use loop scopes in different settings than their current use case, the analysis of **`while`** loops by invariants, for which they proved to be useful and efficient (cf. Sect. 2.3.3).

An approach in the spirit of loop scopes especially addressing **`for`** loops has been proposed in [WS20]. It is based on a new statement type called **`attempt-continuation`**

[10] The reason was the change to IsContinue (see Fig. 2.8), where remaining statements after a **`continue`** were *not* removed. Given the definition of the invariant rule itself and how Java blocks are handled in KeY, this was unproblematic, since remaining statements could only ever be special code to execute the update of a **`for`** loop. However, this is unsound in general.

statement. In this section, we introduce the novel concept of *completion scopes*[11], which facilitate "catching" different reasons of abrupt completion separately, allowing for *specific* reactions. One of the applications of completion scopes is the construction of transparent loop invariant rules for **`while`** and **`for`** loops. In contrast to **`attempt-continuation`** statements, completion scopes are not confined to loops, though. They could also be used to construct *product programs* [BCK11] with abrupt completion. More importantly (for this thesis), we need them in Sect. 4.2 to define the semantics of Abstract Program Elements. Subsequently, we describe syntax, semantics and calculus rules of completion scopes. We leave the implementation of completion scopes to future work.

2.4.1 Syntax and Semantics of Completion Scopes

Completion scope statements are an extension of Java and may legally appear (cf. Sect. 2.2.1) in JavaDL modalities. Syntactically, we align completion scope statements with **`try`** statements extended by "catch" clauses for **`return`**s, **`break`**s, and **`continue`**s. We chose not to extend the *existing* **`try`** statement to not interfere with the original definition of its semantics, and because we do not include **`finally`** blocks.

Definition 2.15 (Completion Scope Statement). A *completion scope statement* is a Java statement adhering to the following grammar:

completionScope ::= "**`exec`**" *block*
("**`ccatch`**" "`(`" "**`\Return`**" "`)`" *block*) ?
("**`ccatch`**" "`(`" "**`\Return`**" *identifier* "`)`" *block*) ?
("**`ccatch`**" "`(`" "**`\Break`**" "`)`" *block*) ?
("**`ccatch`**" "`(`" "**`\Break`**" *identifier* "`)`" *block*) $*$
("**`ccatch`**" "`(`" "**`\Break`**" "`_`" "`)`" *block*) ?
("**`ccatch`**" "`(`" "**`\Continue`**" "`)`" *block*) ?
("**`ccatch`**" "`(`" "**`\Continue`**" *identifier* "`)`" *block*) $*$
("**`ccatch`**" "`(`" "**`\Continue`**" "`_`" "`)`" *block*) ?
("**`ccatch`**" "`(`" *excClassType identifier* "`)`" *block*) $*$

[11] The idea of completion scopes arose in discussions with *Nathan Wasser*, the original inventor of the loop scope concept and co-author of [SW17]. So far, there is no publication on completion scopes; we present them here since we need them. Nathan will be an author of a future publication on the topic.

where *block* is a Java block, *identifier* a valid Java identifier, and *excClassType* an identifier for a Java type which (a subclass of) the type `java.lang.Throwable`. ◊

The keyword **ccatch** stands for "completion catch"; again, we do not re-use the keyword **catch** to avoid confusion with the existing **try** statement. To furthermore emphasize that we are not merely concerned with "failed" computations raising exceptions, but with arbitrary abrupt completion, we use "**exec**" instead of "**try**". We use "completion scope statement" and "**exec** statement" interchangeably. The "**exec** block" of an **exec** statement is the block following the "**exec**" keyword. In the declaration of formal parameters of **ccatch** clauses, **\Break**, **\Continue** and **\Return** are special "types" denoting that the body completed due to a **break**, **continue**, or **return**, respectively. Identifiers after the keywords **\Break** and **\Continue** denote labels passed to labeled **break** and **continue** statements. In contrast to the **ccatch** clauses for **\Return** and *excClassType*, they are no formal parameters which are assigned a dynamic value, but literal identifiers of labels which are *matched* rather than instantiated. A wildcard case for the labeled **\Break** and **\Continue** completion types ("**\Break _**" and "**\Continue _**") can catch *any* labeled **break** and **continue**. Def. 2.15 permits **exec** statements without **ccatch** clauses **exec** { p } (which has the same semantics as p, apart from variable scoping). The order of the abrupt completion types is fixed (for instance, a **\Return** has to be declared before a **\Continue** or an exception **ccatch** clause); this is to ease the definition of calculus rules in the next section. In practice, we would allow arbitrary interleavings of **ccatch** clauses, where, however, the order of different exception **ccatch**s *does* make a semantic difference (as in the case of the **try** statement).

We define the semantics of completion scope statements as below, following, as in the case of loop scopes, the procedural style of the JLS [Gos+05].

Definition 2.16 (Semantics of Completion Scope Statements). An **exec** statement is executed by first executing the **exec** block. Then there is a choice:

- If execution of the **exec** block completes normally, then no further action is taken and the **exec** statement completes normally.
- If execution of the **exec** block completes abruptly because of a **throw** of a value *val*, then there is a choice:
  - If the run-time type of *val* is assignment compatible with a catchable exception class of any **ccatch** clause of the **exec** statement, then the first (leftmost) such **ccatch** clause is selected. The value *val* is assigned to the parameter of the selected **ccatch** clause, and the block of that **ccatch** clause is executed, and then there is a choice:

    * If that block completes normally, then the **exec** statement completes normally.
    * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
  – If the run-time type of *val* is not assignment compatible with a catchable exception class of any **ccatch** clause of the **exec** statement, then the **exec** statement completes abruptly because of a **throw** of the value *val*.

- If execution of the **exec** block completes abruptly because of a **return** of no value, then there is a choice:
  – If the **exec** statement has a **ccatch** clause of type **\Return** with no parameter, then the first (leftmost) such **ccatch** clause is executed, and then there is a choice:
    * If that block completes normally, then the **exec** statement completes normally.
    * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
  – If the **exec** statement has no a **ccatch** clause of type **\Return** with no parameter, then the **exec** statement completes abruptly because of a **return** of no value.
- If execution of the **exec** block completes abruptly because of a **return** of a value *val*, then there is a choice:
  – If the **exec** statement has a **ccatch** clause of type **\Return** with a parameter, then the first (leftmost) such **ccatch** clause is selected. The value *val* is assigned to the parameter of the selected **ccatch** clause, and the block of that **ccatch** clause is executed, and then there is a choice:
    * If that block completes normally, then the **exec** statement completes normally.
    * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
  – If the **exec** statement has no a **ccatch** clause of type **\Return** with a parameter, then the **exec** statement completes abruptly because of a **return** of the value *val*.
- If execution of the **exec** block completes abruptly because of a **continue** with no

label, then there is a choice:

  - If the **exec** statement has a **ccatch** clause of type **\Continue** with no parameter, then the first (leftmost) such **ccatch** clause is executed, and then there is a choice:
    * If that block completes normally, then the **exec** statement completes normally.
    * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
  - If the **exec** statement has no a **ccatch** clause of type **\Continue** with no parameter, then the **exec** statement completes abruptly because of a **continue** with no label.

- If execution of the **exec** block completes abruptly because of a **continue** with label l, then there is a choice:
  - If the **exec** statement has a **ccatch** clause of type **\Continue** with parameter "*l*", then the first (leftmost) such **ccatch** clause is executed, and then there is a choice:
    * If that block completes normally, then the **exec** statement completes normally.
    * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
  - If the **exec** statement has no **ccatch** clause of type **\Continue** with parameter "*l*", but a **ccatch** clause of type **\Continue** with wildcard parameter "**_**", then the first (leftmost) such **ccatch** clause is executed, and then there is a choice:
    * If that block completes normally, then the **exec** statement completes normally.
    * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
  - If the **exec** statement has no a **ccatch** clause of type **\Continue** with parameter "*l*" and also no **ccatch** clause of type **\Continue** with wildcard parameter "**_**", then the **exec** statement completes abruptly because of a **continue** with label l.

- If execution of the **exec** block completes abruptly because of a **break** with no label, then there is a choice:

- If the **exec** statement has a **ccatch** clause of type **\Break** with no parameter, then the first (leftmost) such **ccatch** clause is executed, and then there is a choice:
  * If that block completes normally, then the **exec** statement completes normally.
  * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
- If the **exec** statement has no a **ccatch** clause of type **\Break** with no parameter, then the **exec** statement completes abruptly because of a **break** with no label.

• If execution of the **exec** block completes abruptly because of a **break** with label l, then there is a choice:

- If the **exec** statement has a **ccatch** clause of type **\Break** with parameter "l", then the first (leftmost) such **ccatch** clause is executed, and then there is a choice:
  * If that block completes normally, then the **exec** statement completes normally.
  * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
- If the **exec** statement has no **ccatch** clause of type **\Break** with parameter "l", but a **ccatch** clause of type **\Break** with wildcard parameter "**_**", then the first (leftmost) such **ccatch** clause is executed, and then there is a choice:
  * If that block completes normally, then the **exec** statement completes normally.
  * If that block completes abruptly for any reason, then the **exec** statement completes abruptly for the same reason.
- If the **exec** statement has no a **ccatch** clause of type **\Break** with parameter "l" and also no **ccatch** clause of type **\Break** with wildcard parameter "**_**", then the **exec** statement completes abruptly because of a **break** with label l.

$\lozenge$

The length of Def. 2.16 is due to necessity to cover all possible reasons for abrupt com-

Listing 2.9: Linear Search (v1)

```
i = 0; result = -1;
while (result < 0 && i < arr.length) {
    if (arr[i] == elem)
        result = i;

    i++;
}
return result;
```

Listing 2.10: Linear Search (v2)

```
j = 0;
while (j < arr.length) {
    if (arr[j] == elem)
        return j;

    j++;
}
return -1;
```

pletion, as there might be a **ccatch** clause for each of them. Nevertheless, the definition is quite uniform, and the cases for exceptions are essentially equal to the definition of the semantics of **try** statement in the JLS. Compared to loop scopes (Def. 2.14), there are no predefined variables which are set (such as the loop scope index), and there is no complicated, hard-coded behavior, such as terminating the whole program if the body completes because of a **continue**. Instead, everything is deferred to the implementation of the corresponding **ccatch** clauses. The following example shows how completion scopes could be employed in the construction of *product programs*.

Example 2.5 (Product Programs with Completion Scopes)**.** Consider the two versions of a program performing a linear search through an integer array `arr` depicted in Listings 2.9 and 2.10. Version one records whether a matching value was found in a variable `result` which is initialized to -1 and ends its search once that variable is positive, while version two directly returns after a positive comparison and does therefore not need `result`. To prove the *relational* property that the two program versions are equivalent, or more specifically for this case, that they return the same value, one can construct a *product program* of them. Product programs have been proposed in [BCK11] to reduce relational verification to functional verification. The idea is that to prove a relational property about two programs p_1 and p_2, one proves by standard program verification techniques a postcondition about their *product* p_{prod}. An easy way to construct a product is to sequentially compose them; alternatively, one can construct deeper interleavings leading to better analyzable products. Listing 2.11 shows a product program of version 2 with version 1 (background colors correspond to those in the listing, which indicate the origin of the respective lines in the program). One immediately recognizes a problem with this approach: Not only would a Java compiler reject the program due to the unreachable **return** statement in Line 19; also, this product terminates early because of the **return**s in Lines 6 and 18. This unveils

Listing 2.11: Linear Search (Product)

```
j = 0;
i = 0; result = -1;

while (j < arr.length) {
    if (arr[j] == elem)
        return j;

    j++;
}

while (result < 0 && i < arr.length) {
    if (arr[i] == elem)
        result = i;

    i++;
}

return -1;
return result;
```

Listing 2.12: Linear Search (Product with Completion Scopes)

```
returned_1 = returned_2 = false; retval_1 = retval_2 = 0;

j = 0;
i = 0; result = -1;
exec {
    while (j < arr.length) {
        if (arr[j] == elem)
            return j;

        j++;
    }
} ccatch (\Return val) {
    returned_2 = true; retval_2 = val;
}

exec {
    while (result < 0 && i < arr.length) {
        if (arr[i] == elem)
            result = i;

        i++;
    }
} ccatch (\Return val) {
    returned_1 = true; retval_1 = val;
}

if (!returned_2) {
    exec {
        return -1;
    } ccatch (\Return val) {
        returned_2 = true; retval_2 = val;
    }
}

if (!returned_1) {
    exec {
        return result;
    } ccatch (\Return val) {
        returned_1 = true; retval_1 = val;
    }
}

assert returned_1 && returned_2 && retval_1 == retval_2;
```

a problem of product programs: Without additional measures, it is not possible to construct a product of programs with abrupt completion.[12] As always, one way to solve the problem would be program transformation: Transform the input programs to programs without abrupt completion. As discussed in the context of loop scopes (Sect. 2.3), this conceptually easy idea comes at high cost in the general case, i.e., in the presence of programs with a lot of irregular behavior, and is hard to document and implement correctly.

We propose using *completion scopes* instead. The idea is simple: Wrap each sequential piece of code in the product which corresponds to a sequential piece of code in one of the source programs into an **`exec`** statement, catch abrupt completion with appropriate **`ccatch`** statements and record occurred completion behavior in freshly introduced flags. Listing 2.12 shows the extended product program with completion scopes. In Line 1, two flags recording whether the components returned and two variables storing the returned results are introduced. The sequential pieces of code containing **`return`** statements are each wrapped into an **`exec`** statement catching abrupt completion by a **`return`** with a value, upon which the event is recorded in the new variables. Finally, in Line 43 we added an assertion of the relational postcondition. The result soundly describes the behavior of the component programs in the presence of **`return`** statements—under the assumption that the array accesses do not raise exceptions. In this case, the assertion in Line 43 is reached and proven to hold. The assumption about the absence of exceptions and the **`if`** statements in Lines 27 and 35 indicate that constructing product programs with completion scopes is not trivial either, even though the idea sounds simple. First, one also has to consider different reasons for abrupt completion, especially if knowledge about the internal structure of the composed programs is limited or should not be taken into account. Second, one has to be careful to not enable control flow that was not possible before. Without the conditional in Line 27, for example, `retval_2` would always be set to -1, even if `elem` has been found before. The latter issue does not arise for simple sequential composition without interleavings, i.e., trivial product programs. In this scenario, completion scopes are indeed a very simple and effective measure to perform relational verification of programs with abrupt completion. $\Diamond$

The example demonstrated how completion scopes can be used for relational program verification. Subsequently, we devise Symbolic Execution rules for the **`exec`** statement. This allows using completion scopes within the JavaDL framework.

[12] The programming language of [BCK11] does not have abrupt completion, thus the issue did not occur.

2.4.2 Calculus Rules

To embed **`exec`** statements in the JavaDL calculus, we first declare that openings "**`exec`** { " of **`exec`** statements are part of the non-active prefix π of JavaDL SE rules; **`ccatch`** clauses (and, as usual, closing braces of the **`exec`** block) are part of the postfix ω. We show and explain a representative subset of the calculus rules for the **`exec`** statement.

Exceptions Rule execThrow (Fig. 2.15) corresponds to the JavaDL rule for a **`try`** statement tryCatchFinallyThrow [Ahr+16, Sect. 3.6.7.6]. The schema variable *cs* represents a (possibly empty) sequence of **`ccatch`** clauses. If the active statement is a **`throw`** statement and the first **`ccatch`** clause one for an exception, we distinguish three cases. If the thrown expression is **`null`**, a `NullPointerException` is thrown instead of **`null`**; if the **`ccatch`** clause catches the exception, it is bound to parameter v and the **`ccatch`** clause is executed; finally, if the **`ccatch`** clause does *not* catch the exception, it is removed. In all cases, the remaining code p after the **`throw`** is eliminated. Rule execThrowNoCcatch (Fig. 2.16) also has an existing equivalent: The rule tryFinallyThrow [Ahr+16, Sect. 3.6.7.6]. If there is no **`ccatch`** statement left over, we throw the exception upward. This rule is slightly simpler than tryFinallyThrow because we have no **`finally`** clause.

Elimination of `exec` and Irrelevant `ccatch` Clauses Similar to execThrowNoCcatch, there are rules passing abrupt completion due to **`return`**, (labeled) **`break`** and (labeled) **`continue`** statements upward if there are no **`ccatch`** clauses, only that for those cases, we do not need special treatment of, e.g., **`null`** values. Rule execEmpty (Fig. 2.17) eliminates an *empty* **`exec`** statement including all **`ccatch`** clauses. A couple of rules eliminate **`ccatch`** clauses not matching the abrupt completion behavior triggered by the active statement. For instance, execThrowReturnCcatch (Fig. 2.21) removes a leading **`ccatch`** clause catching abrupt completion due to a **`return`** of no value when the active statement is a **`throw`** statement; execBreakNonMatchingLabel (Fig. 2.20) removes a **`ccatch`** clause for a labeled break with a label not matching the label of the active labeled **`break`**. Additionally, there are more rules discarding **`ccatch`** clauses in related situations.

Rules for `return`, `break`, `continue` A return of a value in a situation where the first **`ccatch`** clause catches this behavior is shown in execReturn (Fig. 2.18). It works as the second case for **`throw`** discussed above: We bind the returned value to parameter v and execute the code q from the **`ccatch`** clause. Rule execBreakLabel (Fig. 2.19) deals with labeled breaks if the leading **`ccatch`** clause matches. The rule is simple: If the label in the execThrow

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{if}}\ (se\texttt{==}\mathtt{\textbf{null}})\ \{\\ \qquad \mathtt{\textbf{exec}}\ \{\ \mathtt{\textbf{throw}}\ \mathtt{\textbf{new}}\ \texttt{NullPointerException();}\ \}\\ \qquad \mathtt{\textbf{ccatch}}\ (T\ v)\ \{\ q\ \}\ cs\\ \ \}\ \mathtt{\textbf{else}}\ \mathtt{\textbf{if}}\ (se\ \mathtt{\textbf{instanceof}}\ T)\ \{\\ \qquad T\ v\texttt{;}\ v\ \texttt{=}\ (T)se\texttt{;}\ q\\ \ \}\ \mathtt{\textbf{else}}\ \{\\ \qquad \mathtt{\textbf{exec}}\ \{\ \mathtt{\textbf{throw}}\ se\texttt{;}\ \}\ cs\\ \ \}\ \omega]\varphi, \Delta\end{array}}{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{exec}}\ \{\ \mathtt{\textbf{throw}}\ se\texttt{;}\ p\ \}\\ \qquad \mathtt{\textbf{ccatch}}\ (T\ v)\ \{\ q\ \}\ cs\ \omega]\varphi, \Delta\end{array}}$$

Figure 2.15: **exec** Rule for **throw**

execThrowNoCcatch

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{if}}\ (se\texttt{==}\mathtt{\textbf{null}})\ \{\\ \qquad \mathtt{\textbf{throw}}\ \mathtt{\textbf{new}}\ \texttt{NullPointerException();}\\ \ \}\ \mathtt{\textbf{else}}\ \{\\ \qquad \mathtt{\textbf{throw}}\ se\texttt{;}\\ \ \}\ \omega]\varphi, \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\textbf{exec}}\ \{\ \mathtt{\textbf{throw}}\ se\texttt{;}\ p\ \}\ \omega]\varphi, \Delta}$$

Figure 2.16: **exec** Rule for **throw** without **ccatch**

ccatch clause *literally* equals the label of the **break**, we execute the code of the **ccatch** clause. There is a similar rule for labeled **continue** statements as well as **break**s and **continue**s without labels (which are even simpler, since no label has to be matched).

Allowing **exec** statements without **ccatch** clauses facilitates eliminating leading non-matching **ccatch** clauses step by step without having to check whether there are any left. The substantial number of SE rules for the **exec** statement reflects the many cases in the definition of the statement's semantics (Def. 2.16). However, almost all rules are very simple and uniform, and therefore easy to construct; the most complex one (for **throw**) is almost identical to an already existing JavaDL rule.

Completion scopes can be seen as a generalization of **try** statements, built on the idea of loop scopes, to catch abrupt completion. Based on this new concept, one could construct new loop invariant rules for **while** loops and also, without prior program transformation, execEmpty

$$\frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \textbf{exec}\ \{\ \}\ cs\ \ \omega]\varphi, \Delta}$$

Figure 2.17: **exec** Rule for Empty **exec** Block

execReturn

$$\frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ T\ \ v;\ \ v\ =\ (T)se;\ \ q\ \ \omega]\varphi, \Delta}{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}[\pi\ \textbf{exec}\ \{\ \textbf{return}\ se;\ \ p\ \ \}\\ \qquad\qquad \textbf{ccatch}\ (\backslash\textbf{Return}\ \ v)\ \ \{\ \ q\ \ \}\ cs\ \omega]\varphi, \Delta\end{array}}\ (*)$$

$(*)$ T is the method return type

Figure 2.18: **exec** Rule for **return** of a Value

execBreakLabel

$$\frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ q\ \ \omega]\varphi, \Delta}{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}[\pi\ \textbf{exec}\ \{\ \textbf{break}\ \ l;\ \ p\ \ \}\\ \qquad\qquad \textbf{ccatch}\ (\backslash\textbf{Break}\ \ l)\ \ \{\ \ q\ \ \}\ cs\ \omega]\varphi, \Delta\end{array}}$$

Figure 2.19: **exec** Rule for **break** with a Matching Label

execBreakNonMatchingLabel

$$\frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \textbf{exec}\ \ \{\ \textbf{break}\ \ l;\ \ \}\ \ cs\ \ \omega]\varphi, \Delta}{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \textbf{exec}\ \ \{\ \textbf{break}\ \ l;\ \ p\ \ \}\\ \qquad\qquad \textbf{ccatch}\ (\backslash\textbf{Break}\ \ l')\ \ \{\ \ q\ \ \}\ \ cs\ \ \omega]\varphi, \Delta\end{array}}\ l \neq l'$$

Figure 2.20: **exec** Rule for **break**, Non-Matching Label in **ccatch** Clause

execThrowReturnCcatch

$$\frac{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \textbf{exec}\ \ \{\ \textbf{throw}\ \ se;\ \ \}\ \ cs\ \ \omega]\varphi, \Delta}{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \textbf{exec}\ \ \{\ \textbf{throw}\ \ se;\ \ p\ \ \}\\ \qquad\qquad \textbf{ccatch}\ (\backslash\textbf{Return})\ \ \{\ \ q\ \ \}\ \ cs\ \ \omega]\varphi, \Delta\end{array}}$$

Figure 2.21: **exec** Rule for **break**, Wrong **ccatch** Clause with a **return**

directly for **for** loops. We defer this task, and also the implementation of completion scopes in KeY, to future work, since it is not in the scope of this thesis. Here, completion scopes are used in Sect. 4.2 to characterize the semantics of abstract program elements and to check whether concrete programs are legal instantiations of those.

2.5 Second-Order Java Dynamic Logic

JavaDL is a program logic for reasoning over first-order properties of individual Java programs. To concisely illustrate the expressivity of AE, our analysis technique for reasoning about properties of *many* programs, we extend JavaDL to JavaDL$^{\mathrm{II}}$, a *second-order* program logic which allows quantifying over *programs*. To that end, we add to the logic's alphabet a set of higher-order variables ranging over program *statements* and *expressions*. This allows us to formally express properties like "there is a program p such that for all normally terminating programs q, the composed program "$q\ p$" terminates in a state where variable `x` has the value 1". In JavaDL$^{\mathrm{II}}$, this could be expressed as

$$\exists^{\mathrm{II}} p : \mathit{Stmt}; \forall^{\mathrm{II}} q : \mathit{Stmt}; (\langle q \rangle \mathrm{true} \rightarrow \langle q\quad p \rangle (\mathtt{x} \doteq 1))$$

The quantifiers $\exists^{\mathrm{II}}$, $\forall^{\mathrm{II}}$ are second-order program quantifiers. The property could be proven, e.g., by instantiating statement variable p with `x=1;` and proving by structural induction over q that if q completes normally, p is executed and `x` will attain the value 1.

We define syntax and semantics of JavaDL$^{\mathrm{II}}$. Afterward, we briefly discuss a reasoning system for the logic.

2.5.1 Syntax and Semantics

Types We extend JavaDL type hierarchies to include the types *Stmt* of Java statements and *Expr* of Java expressions. The two types intersect, since there are expressions, the so-called expression statements, that are also statements (e.g., `x++`).

Signatures We extend JavaDL signatures by sets

- ProgVSym of typed rigid second-order variable symbols over Java statements and expressions, where by writing $p : A$ for $p \in$ ProgVSym we declare p to be a variable of type $A \in \{\mathit{Stmt}, \mathit{Expr}\}$,

- ProgCSym of typed second-order constant symbols over Java statements and expressions, where by writing $c : A$ for $c \in \text{ProgCSym}$ we declare c to be a constant of type $A \in \{\mathit{Stmt}, \mathit{Expr}\}$.

A JavaDL$^{\text{II}}$ signature is thus a tuple

$$\Sigma = (\text{FSym}, \text{PSym}, \text{VSym}, \text{PVSym}, \text{ProgVSym}, \text{ProgCSym})$$

Terms and Formulas The set Trm_A^{II} of JavaDL$^{\text{II}}$ terms is as in JavaDL. For formulas, we first extend the notion of legal program fragments defined in Sect. 2.2.1 by allowing second-order statement and expression variables and constants to appear inside modalities wherever any concrete Java statement or expression might appear. Additionally, the expressions $\forall^{\text{II}} v; \varphi$ and $\exists^{\text{II}} v; \varphi$ are in the set Fml^{II} of JavaDL$^{\text{II}}$ formulas for formulas $\varphi \in \text{Fml}^{\text{II}}$ and $v \in \text{ProgVSym}$. The notion of closed formulas extends to variables $v \in \text{ProgVSym}$. As for JavaDL terms, we call a Java statement / expression *ground* if it does not contain second-order variables.

Semantics To evaluate JavaDL$^{\text{II}}$ formulas, we extend JavaDL Kripke structures to tuples $K = (\mathscr{D}, \delta, I, I^2, \mathscr{S}, \varrho)$ additionally containing an interpretation function I^2 assigning concrete Java programs to second-order program constants in ProgCSym. Variable assignments β also range over second-order program variables in ProgVSym. Then, we define the valuation function val^2 of JavaDL$^{\text{II}}$ as follows:

Definition 2.17 (JavaDL$^{\text{II}}$ Semantics)**.** Let *Prg* be a Java program (without second-order program variables or constants), $\mathscr{T}$ be a type hierarchy for *Prg*, Σ a signature w.r.t. $\mathscr{T}$, $K = \left(\mathscr{D}, \delta, I, I^2, \mathscr{S}, \varrho\right)$ a second-order Kripke structure for Σ, $\sigma \in \mathscr{S}$ a state, and β a variable assignment defined on $(\text{VSym} \cup \text{ProgVSym})$. The valuation function $\mathit{val}^2(K, \sigma, \beta|\cdot)$ is defined as the JavaDL $\mathit{val}(K, \sigma, \beta|\cdot)$ (Def. 2.8) except for modalities and second-order quantifiers, where we define it as depicted in Fig. 2.22. It is a *partial* function: If replacing second-order variables and constants in modalities according to their assignments in I^2 and β leads to an illegal program fragment, the function is not defined. For convenience, we use the notation $\mathit{val}^2(K, \sigma, \beta|\cdot) = \mathit{nn}$ for undefinedness in the figure. $\Diamond$

2.5.2 Reasoning

For reasoning in JavaDL$^{\text{II}}$, we introduce five proof rules: Two of them, sndAllRight and sndExLeft, introduce fresh second-order Skolem constants for universally quantified

$$val^2(K,\sigma,\beta|[p]\varphi) \overset{(\star)}{=} \begin{cases} nn & \text{if } p' \text{ is not a legal program fragment} \\ tt & \text{if } p' \text{ is a legal program fragment and} \\ & val^2\big(K,\sigma,\beta|[p']\varphi\big) = tt \\ ff & \text{otherwise} \end{cases}$$

$$val^2(K,\sigma,\beta|\langle p\rangle\varphi) \overset{(\star)}{=} \begin{cases} nn & \text{if } p' \text{ is not a legal program fragment} \\ tt & \text{if } p' \text{ is a legal program fragment and} \\ & val^2\big(K,\sigma,\beta|\langle p'\rangle\varphi\big) = tt \\ ff & \text{otherwise} \end{cases}$$

$$val^2\big(K,\sigma,\beta|\forall^{\mathrm{II}}v : A;\varphi\big) = \begin{cases} tt & \text{if there is no Java construct } p \text{ of type } A \text{ s.t.} \\ & val^2(K,\sigma,\beta[v \mapsto p]|\varphi) = ff \\ ff & \text{otherwise} \end{cases}$$

$$val^2\big(K,\sigma,\beta|\exists^{\mathrm{II}}v : A;\varphi\big) = \begin{cases} tt & \text{if for at least one Java construct } p \text{ of type } A \text{ it} \\ & \text{holds that } val^2(K,\sigma,\beta[v \mapsto p]|\varphi) = tt \\ ff & \text{otherwise} \end{cases}$$

$$\vdots$$

($\star$) where p' results from p by substituting all second-order program constants by their interpretations in I^2 and second-order program variables by their assignments in β.

Figure 2.22: JavaDL$^{\mathrm{II}}$ Semantics

second-order variables in the succedent and existentially quantified second-order variables in the antecedent. Another two rules, sndExRight and sndAllLeft, allow the substitution of concrete Java statements and expressions for existentially quantified second-order variables in the succedent and universally quantified variables in the antecedent, as long as the result is a legal program fragment. These rules, shown in Fig. 2.23, resemble the quantifier rules of first-order logic or JavaDL. However, the Skolemization rules are far too weak to show most interesting properties. Consider, e.g., the example from above:

$$\vdash \exists^{\mathrm{II}}p : Stmt; \forall^{\mathrm{II}}q : Stmt; (\langle q\rangle\,\mathrm{true} \rightarrow \langle q\ \ p\rangle(\mathtt{x} \doteq 1))$$

After instantiating the variable p with the sensible choice of `x=1;`, all we can do is introducing a fresh second-order Skolem variable for q, resulting in the sequent

$$\text{sndAllRight}\ \frac{\Gamma \vdash \varphi[c/v], \Delta}{\Gamma \vdash \forall^{\mathrm{II}} v : A; \varphi, \Delta}$$
with $c : A$ a new constant

$$\text{sndExLeft}\ \frac{\Gamma, \varphi[c/v] \vdash \Delta}{\Gamma, \exists^{\mathrm{II}} v : A; \varphi \vdash \Delta}$$
with $c : A$ a new constant

$$\text{sndExRight}\ \frac{\Gamma \vdash \varphi[p/v], \Delta}{\Gamma \vdash \exists^{\mathrm{II}} v : A; \varphi, \Delta}$$
with $p : A$ a ground Java program s.t. programs inside modalities in $\varphi[p/v]$ are legal program fragments

$$\text{sndAllLeft}\ \frac{\Gamma, \varphi[p/v] \vdash \Delta}{\Gamma, \forall^{\mathrm{II}} v : A; \varphi \vdash \Delta}$$
with $p : A$ a ground Java program s.t. programs inside modalities in $\varphi[p/v]$ are legal program fragments

Figure 2.23: Second-Order Quantifier Rules of JavaDL$^{\mathrm{II}}$

$$\vdash \langle q_0 \rangle \,\mathrm{true} \rightarrow \langle q_0 \;\; \texttt{x=1;} \rangle (\texttt{x} \doteq 1)$$

From here, we cannot proceed, since there are no symbolic execution rules for second-order Skolem constants. The authors of [BRR08] (which we discuss in Chapter 7) propose to “decompose” q_0 into a normally completing part with side effects and a side effect-free, but abruptly completing part, and to then split the modality (“linearization”). Abstract Execution essentially addresses this situation, too, but uses *abstract updates* instead of decomposition, which behaves well with JavaDL prefixes π. Furthermore, AE provides a mechanism to automatically create assumptions like “$\langle q_0 \rangle$ true”. The best-known approach to proving second-order program properties is *structural induction* [Bur74]. Based on an inductive definition of the Java language, a structural induction rule structIndAllRight considers all language constructs in separate proof cases, with corresponding hypotheses for non-basic ones. In our example, the case for an **if** statement would look like

$$\begin{aligned}
\vdash &\big(((\langle q_1 \rangle \,\mathrm{true}) \rightarrow \langle q_1 \;\; \texttt{x=1;} \rangle (\texttt{x} \doteq 1)) \land \\
&\quad ((\langle q_2 \rangle \,\mathrm{true}) \rightarrow \langle q_2 \;\; \texttt{x=1;} \rangle (\texttt{x} \doteq 1))\big) \rightarrow \\
&\qquad \big((\langle \textbf{if}\ (b)\ q_1\ \textbf{else}\ q_2 \rangle \,\mathrm{true}) \rightarrow \\
&\qquad\qquad \langle \textbf{if}\ (b)\ q_1\ \textbf{else}\ q_2 \;\; \texttt{x=1;} \rangle (\texttt{x} \doteq 1)\big)
\end{aligned}$$

Standard SE rules for the **if** statement reduce the formula in the conclusion to one of the base cases in the premise, thus allowing to close this proof branch. Due to the extent of the Java language, structIndAllRight creates many branches. Even though most, if not all, of them should be provable automatically, this amounts to a considerable proof effort.

The possibility to perform structural induction to prove universal second-order properties in our logic indicates that our "second-order" dynamic logic is much weaker than classical second-order logic with quantification about *arbitrary sets*; the set of Java constructs over which our quantifiers range has much more structure.

3 A Theory of Symbolic Execution

Symbolic Execution (SE) [Bal+18; Yan+19] is a popular program analysis technique introduced in the 1970s [Bur74; Kin76; DE82] for exploring a large number of execution paths of a program. The key idea is to treat inputs to a program as abstract symbols. Whenever execution depends on the concrete value of a symbolic variable, SE follows the different execution paths simultaneously. Symbolic Execution engines maintain for each explored path (1) a *path condition* describing the conditions satisfied by the branches taken along that path, and (2) a *symbolic store* mapping variables to (symbolic) values, (3) a *program counter* pointing to the next instruction to execute. Branch execution updates the path condition, while assignments update the symbolic store [Bal+18]. The triple consisting of these elements is called a *Symbolic Execution State (SES)*. Representations of symbolic configurations may vary; examples are Symbolic Execution Trees (SETs), directed acyclic graphs or simply sets of states. We distinguish two types of SE:

(1) *Lightweight* SE has its applications in bug finding or program testing. These approaches, while exploring more program paths than concrete execution, typically underapproximate the set of all theoretically possible paths. Frequently, concrete and symbolic execution are mixed (dubbed "concolic" execution, for "*conc*rete" and "symb*olic*"; see, e.g., [JMN13; Jam+12; CKC12]). Loops and recursive methods are treated by unwinding up to a predefined upper bound: In general, the cost of exhaustive program exploration is kept under control by only exploring heuristically selected paths (e.g., [CDE08; Ma+11; CKC12]). Practically, analyzed programs are frequently instrumented by replacing data types with symbolic representations or by adding function calls to the SE engine, which in turn is backed by an external Satisfiability Modulo Theories (SMT) solver. Lightweight SE has been employed in the analysis of whole software libraries [CDE08]. Example systems include KLEE [CDE08], Java PathFinder [PV04] and S2E [CKC12].

(2) *Heavyweight* SE overapproximates the set of possible paths through a program and thus can be used to *prove* complex functional properties about programs. At branching points, all possible paths are followed independently; loops with symbolic

guards and recursive methods are treated by abstraction, for instance by using loop invariants. A strong focus is put on *modularity*: e.g., single methods may be thoroughly analyzed independently from the concrete code of others. To achieve this, the analysis depends on specifications such as *method contracts* or *class invariants*. Heavyweight SE systems execute analyzed programs with a dedicated symbolic interpreter. While doing so, they can rely on an external solver, or be integrated with an internal theorem proving engine. Due to high computation time and the effort required for creating specifications, they do not scale to complete libraries, and are instead used to assert strong guarantees about critical routines [GBR14; Bec+17; Gou+19] or to build powerful tools like symbolic debuggers [HHB14]. Example systems encompass KeY [Ahr+16], VeriFast [VJP15] and KIV [Ste05].

In this thesis, we focus on *heavyweight* SE, since our focus is to rigorously prove properties, for which it is necessary to explore (at least) all practically possible program paths. The concepts introduced in the remainder of this section (semantics of SESs, SE transition relations, exhaustiveness/precision of SE transition relations, and state merging) are, however, universally applicable.

Considering the popularity and usefulness of SE, it is surprising that literature on the *semantic foundations* of the technique is very rare. Surveys like [Bal+18; Yan+19] precisely define the *syntactic* aspects, like the grammar to construct SESs, but then directly move on to technical details (like implementation strategies for SE engines or solution strategies to practical problems). Individual papers frequently apply SE without going much into details, assuming that the concepts are familiar enough to the reader. In the KeY framework [Ahr+16], SE is tightly integrated into the program logic. An SE rule is simply some JavaDL rule changing the content of a modality; the correctness notion of SE rules boils down to a special case of the correctness of JavaDL sequent calculus rules. There is no explicit notion of an SE state or a semantics for SE.

We know only three approaches to defining a general theoretic framework for SE [Kne91; LRA17; BB19]. The more recent definitions by Lucanu et al. [LRA17] and de Boer and Bonsangue [BB19] choose an approach relating symbolic and concrete execution by simulation relations; they do not consider the semantics of SESs. We refer to the discussion in Chapter 7 for a closer description and comparison to our work. The earliest work on the semantics of SE, which is also closest in spirit to out framework, is Kneuper's paper [Kne91] from the early '90s. The author defines the "denotational semantics of symbolic execution of specifications and programs", and aims to thus provide a general correctness notion. Two versions of an SE semantics are defined: *Full* SE which precisely captures the set of all possible execution paths, and *weak* SE which overapproximates this set. The latter is frequently required in programs with loops or recursive methods, where invariants or method summaries are needed as auxiliary specifications. In this chapter, we provide a general theory of Symbolic Execution built upon three pillars: (1) A denotational semantics, embedded into JavaDL, of SESs based on *concretizations* to concrete states, (2) a general definition of SE transition relations permitting not only "1-to-n", but "m-to-n" transitions, and (3) two definitions of *precision* and *exhaustiveness* of SE transition relations and a discussion of their properties and relation to lightweight and heavyweight SE. Compared to [Kne91], whose definitions of full and weak SE constitute important preparatory work, our framework is more general, since it permits state merging, and more concise. Moreover, our correctness notions are more modular: *Full* SE is *both* precise *and* exhaustive. Weak SE corresponds to exhaustive SE.

In Sect. 3.1, we define the syntax and semantics of Symbolic Execution States. These definitions are loosely based on our earlier work in [SHB16; SH18; SH19b]. Afterward, we define m-to-n SE transition relations and their properties of precision and exhaustiveness in Sect. 3.2. Sect. 3.3 concludes the chapter with a discussion of state merging.

3.1 Syntax and Semantics of Symbolic Execution States

As usual, our SESs are a triples of a path condition, a symbolic store, and a program counter. The order of these elements occasionally varies in literature; e.g., in [BB19], it is reversed. We represent path conditions by closed formulas and symbolic stores by JavaDL *updates*. Updates have the advantage that we can evaluate, e.g., a formula, in a symbolic store by simply *applying* the update to the formula. A program counter in our framework is the whole remaining program (instead of a pointer to the next instruction).

Definition 3.1 (Symbolic Execution State). A *Symbolic Execution State (SES)* is a triple $(C, \mathcal{U}, p)$ of (1) a *path condition*, formalized as a set of *closed* formulas $C \in 2^{\mathrm{Fml}}$, (2) a *symbolic store*, formalized as an update $\mathcal{U} \in \mathrm{Upd}$, and (3) a *program counter*, formally a *legal (Java) program fragment* (cf. Sect. 2.2.1) p. We write $(C, \mathcal{U})$ for SESs with empty program counters, and denote the set of all SESs by $\mathbb{S}_{SE}$. $\Diamond$

Semantically, a *symbolic* execution state $s \in \mathbb{S}_{SE}$ represents a (potentially infinite) set of *concrete* execution states $\sigma \in \mathcal{S}$, in the same way as a symbolic parameter represents an up to infinite set of concrete parameters. We call this set of concrete states *concretizations* for s. Based on the valuation function of JavaDL, we define the *concretization function* *concr* which, given an initial concrete state σ, concretizes a symbolic state s to a concrete state. The union $\bigcup_\sigma concr(s,\sigma)$ for all initial states represents the set of concretizations. We begin with a "K-indexed" version $concr_K$ and then define $concr$ as the union for all structures K. The idea is that all different interpretations of uninterpreted function and predicate symbols are captured in the concretizations. If, for instance, new Skolem symbols are introduced after a loop invariant application, the represented concrete state space is extended, which must be reflected in the definition. In this thesis, we assume, for simplifying the presentation, that all Kripke structures K have the same signature which is adequate for the context in which they are used. This implies that introducing a "new" Skolem symbol means to use a symbol which already exists in the signature, but is not yet used present in the context, e.g., in the current SES.

Definition 3.2 ((K-indexed) Concretization Function)**.** The *K-indexed concretization function* $concr_K : \mathbb{S}_{SE} \times \mathcal{S} \to 2^{\mathcal{S}}$ maps an SES $(C,\mathcal{U},p)$ and a concrete state $\sigma \in \mathcal{S}$ (1) to the empty set $\emptyset$ if *either* $K,\sigma \not\models \bigwedge C$, *or*, where $\sigma' := val(K,\sigma|\mathcal{U})(\sigma)$, there is no σ'' s.t. $(\sigma',\sigma'') \in \varrho(p)$, or otherwise (2) to the singleton set $\{\sigma''\}$ s.t. $(\sigma',\sigma'') \in \varrho(p)$, where σ' is as before. The *concretization function concr* is defined as $concr(s,\sigma) := \bigcup_K concr_K(s,\sigma)$. $\Diamond$

Definition 3.3 (Semantics of SESs)**.** The semantics $[\![s]\!]$ of an SES $s \in \mathbb{S}_{SE}$ is defined as the union of its concretizations: $[\![s]\!] := \bigcup_{\sigma\in\mathcal{S}} concr(s,\sigma)$. $\Diamond$

We illustrate Defs. 3.2 and 3.3 along two examples: one for "full" and one for "weak" SE, following the terminology of [Kne91].

Example 3.1 (Concretization of SESs)**.** Consider the program

```
y = x; if (x<0) x=-x;
```

The SES $s = \big(\emptyset, \mathtt{y} := \mathtt{x}, \texttt{\textbf{if} (x<0) x=-x;}\big)$ is an intermediate result of an SE engine for this program after symbolic execution of the assignment. Since so far, no branches have been taken, the path condition is still empty; the program variable `y` has been assigned the initial value of program variable `x`, and the conditional statement inverting `x` if it is negative is scheduled to be executed next. Thus, the state represents all concrete states where `x` is positive and `y` equals either `x` or its inverse. Given, for instance, the initial concrete state σ with $\sigma(\mathtt{x}) = -17$ and $\sigma(\mathtt{y}) = 21$, we have

$$\begin{aligned} concr(s,\sigma) &= \bigcup_K \{\sigma'' : (val(K,\sigma|\mathtt{y} := \mathtt{x})(\sigma),\sigma'') \in \varrho(\texttt{\textbf{if} (x<0) x=-x;})\} \\ &= \{\sigma'' : (\sigma[\mathtt{y} \mapsto -17],\sigma'') \in \varrho(\texttt{\textbf{if} (x<0) x=-x;})\} \\ &= \{\sigma[\mathtt{y} \mapsto -17][\mathtt{x} \mapsto 17]\} \end{aligned}$$

since $K,\sigma \models \bigwedge \emptyset$ (which is equivalent to $K,\sigma \models \text{true}$) for any K and the program counter terminates (for all initial states, thus also for $\sigma[\mathtt{y} \mapsto -17]$). The semantics $[\![s]\!]$ of s is the union for all initial states σ:

$$\begin{aligned} \bigcup_{\sigma\in\mathscr{S}} concr(s,\sigma) &= \bigcup_{\sigma\in\mathscr{S}} \bigcup_K \{\sigma'' : (val(K,\sigma|\mathtt{y} := \mathtt{x})(\sigma),\sigma'') \in \varrho(\texttt{\textbf{if} (x<0) x=-x;})\} \\ &= \bigcup_{\sigma\in\mathscr{S}} \{\sigma'' : (\sigma[\mathtt{y} \mapsto \sigma(\mathtt{x})],\sigma'') \in \varrho(\texttt{\textbf{if} (x<0) x=-x;})\} \\ &= \bigcup_{\sigma\in\mathscr{S}} \{\sigma[\mathtt{y} \mapsto \sigma(\mathtt{x})][\mathtt{x} \mapsto |\sigma(\mathtt{x})|]\} \end{aligned}$$

Continuing execution from s yields two successor states, one for positive initial values of `x`, where the end of the program is reached, and one for negative initial values, where `x` is inverted before the end of the program. Those states have nonempty path conditions:

$$\begin{aligned} s_1 &= (\{\mathtt{x} \geq 0\}, \mathtt{y} := \mathtt{x}) \\ s_2 &= (\{\mathtt{x} < 0\}, \mathtt{y} := \mathtt{x}, \texttt{x=-x;}) \end{aligned}$$

For the initial state σ from above, $concr(s_1,\sigma)$ is $\emptyset$, since the path condition of s_1 is not satisfied by σ: $K,\sigma \not\models \mathtt{x} \geq 0$. The concretization of s_2 equals the one for s, since the path condition of s_2 is satisfied by σ. We have $concr(s,\sigma) = concr(s_1,\sigma) \cup concr(s_2,\sigma)$. ◊

The following example demonstrates the concept of Def. 3.2 for "weak" SE, where a loop with a symbolic guard is abstracted by a loop invariant.

Example 3.2 (Concretization of SESs with Skolem Constants). We consider a program which decrements a positive variable inside a loop until it reaches 0 and adds 2 afterward:

```
while (i > 0) { i--; } i += 2;
```

Since the initial value of `i`, and therefore the number of loop iterations, is unknown, we have to abstract the loop by an invariant. Let p be the above program. SE starts with the initial SES $s = (\{\mathtt{i} \geq 0\}, Skip, p)$, where the path condition contains the precondition that $\mathtt{i}$ is nonnegative. A suitable inductive loop invariant for the **while** loop in p is $\mathtt{i} \geq 0$. Together with the negated loop guard $\mathtt{i} \leq 0$, which holds *after* termination of the loop, this is sufficiently strong to infer that $\mathtt{i}$ is actually 0 after loop termination. From the application of a loop invariant rule, in which we, as a side condition, first would have to show that $\mathtt{i} \geq 0$ is actually a loop invariant, we obtain the successor state $s_1 = (\{c \geq 0, c \leq 0\}, \mathtt{i} := c, \mathtt{i\ \ +=\ \ 2;})$, where $\mathtt{i} := c$ is an anonymizing update with a Skolem constant c, $c \geq 0$ the loop invariant, and $c \leq 0$ the branch condition signifying that the loop has been exited. Semantic-preserving simplification of the path condition leads to the state $s_1' = \big(\{c \doteq 0\}, \mathtt{i} := c, \mathtt{i\ \ +=\ \ 2;}\big)$. Since it contains an uninterpreted constant c, the parameter K of the K-indexed concretization function $concr_K$ is important (compared to Example 3.1 where it did not make a difference):

$$
\begin{aligned}
[\![s_1']\!] = \bigcup_{\sigma} concr(s_1', \sigma) &= \\
&\bigcup_{\sigma \in \mathscr{S}} \bigcup_{K} \big\{\sigma' : (val\big(K, \sigma | \mathtt{i} := c\big)(\sigma), \sigma') \in \varrho(\mathtt{i\ \ +=\ \ 2;}) \text{ and } K, \sigma \models c \doteq 0\big\} \\
\overset{(*)}{=} \quad &\bigcup_{\sigma \in \mathscr{S}} \bigcup_{K} \big\{\sigma' : (val\big(K, \sigma | \mathtt{i} := 0\big)(\sigma), \sigma') \in \varrho(\mathtt{i\ \ +=\ \ 2;})\big\} \\
= \quad &\bigcup_{\sigma \in \mathscr{S}} \big\{\sigma' : (\sigma[\mathtt{i} \mapsto 0], \sigma') \in \varrho(\mathtt{i\ \ +=\ \ 2;})\big\} \\
= \quad &\bigcup_{\sigma \in \mathscr{S}} \big\{\sigma[\mathtt{i} \mapsto 2]\big\}
\end{aligned}
$$

Consequently, s_1' represents all concrete states where $\mathtt{i}$ attains the value 2. Step $(*)$ results from the following considerations: If all structures K in the specified set are such that $K, \sigma \models c \doteq 0$, then the transformers created for $val(K, \sigma | \mathtt{i} := c)$ are equivalent to those created for $val(K, \sigma | \mathtt{i} := 0)$. After this simplification, there remain no more rigid symbols, and the union over all K can be omitted.

For any number $k \leq 0$, the formula $\mathtt{i} \geq k$ is also a valid loop invariant. If k is *strictly* negative, the SES resulting after executing the loop has more than one concretization. If we choose $k := -1$, for example, $\mathtt{i}$ can attain the values -1 *or* 0 after the loop. Consequently, concretizations for the final SES after the program comprise some where $\mathtt{i}$ is increased by two, and some where it is only increased by one. This is characteristic for *weak* SE: We compute an *overapproximation*. $\Diamond$

SE is frequently used to derive *weakest preconditions* for program proving. Consider, e.g., the program $p := \mathtt{x=1;}$ and the postcondition $\varphi := \mathtt{x} > \mathtt{y}$. The weakest precondition for p w.r.t. φ is $1 > \mathtt{y}$: Whenever we may assume that $\mathtt{y}$ is strictly smaller than 1, p satisfies φ.

We label SESs with postconditions to denote their weakest precondition w.r.t. the given postcondition, and call s^{φ} a *labeled SES*.

Definition 3.4 (Labeled Symbolic Execution State). Let $s = (C, \mathscr{U}, p) \in \mathbb{S}_{SE}$ and $\varphi \in \mathrm{Fml}$. We write s^{φ} for the *Labeled Symbolic Execution State* s with postcondition φ. We define the semantics of labeled SESs as follows:

$$val(K, \sigma | s^{\varphi}) := \begin{cases} tt & \text{if for all } \sigma' \in concr_K(s, \sigma),\ val\big(K, \sigma' | \varphi\big) = tt \\ ff & \text{otherwise} \end{cases} \qquad \Diamond$$

Since labeled SESs evaluate to truth values, we can use them as formulas, e.g., in semantic entailments: For instance, $\psi \models s^{\varphi}$ holds if ψ implies the weakest precondition of s with respect to postcondition φ. If ψ can be omitted, we call a labeled SES *valid*.

Example 3.3 (Validity of Labeled SESs). For the SES $s = \big(\emptyset, \mathtt{y} := \mathtt{x}, \mathtt{\textbf{if}\ (x<0)\ x=-x;}\big)$ of Example 3.1, the postcondition $\mathtt{x} \geq 0$ holds, i.e., $\models s^{\mathtt{x}\geq 0}$: Let K, σ be a structure and concrete state. From Example 3.1, we know that

$$concr_K(s, \sigma) = \{\sigma[\mathtt{y} \mapsto \sigma(\mathtt{x})][\mathtt{x} \mapsto |\sigma(\mathtt{x})|]\}$$

For $K, \sigma \models s^{\mathtt{x}\geq 0}$ to hold, by Def. 3.4 we therefore need to show, for

$$\sigma' := \sigma[\mathtt{y} \mapsto \sigma(\mathtt{x})][\mathtt{x} \mapsto |\sigma(\mathtt{x})|],$$

that $K, \sigma' \models \mathtt{x} \geq 0$, which follows from $\sigma'(\mathtt{x}) \geq 0$. The postcondition $\mathtt{y} \geq 0$ does not necessarily hold since $\sigma'(\mathtt{y}) = \sigma(\mathtt{x})$ can be negative. $\Diamond$

Example 3.4 (Labeled SESs for Invariant Reasoning). We connect to Example 3.2. For the proof that the loop preserves the invariant $\mathtt{i} \geq 0$, one has to show the validity of the labeled SES $(s_2')^{\mathtt{i}\geq 0}$, where $s_2' = (\{c > 0\}, \mathtt{i} := c, \mathtt{i--;})$ is obtained from $s_2 = (\{c \geq 0, c > 0\}, \mathtt{i} := c, \mathtt{i--;})$ by simplification of the path condition (the formula $c > 0$ is the branch condition signalling that the loop has not yet been left). The path condition,

$c > 0$, of s_2' is an *in*equation and thus more abstract than the one of s_1':

$$
\begin{aligned}
[\![s_2']\!] = \bigcup_{\sigma} concr(s_2', \sigma) = \\
&= \bigcup_{\sigma \in \mathcal{S}} \bigcup_{K} \left\{\sigma' : \left(val\left(K, \sigma | \mathtt{i} := c\right)(\sigma), \sigma'\right) \in \varrho(\mathtt{i--;}) \text{ and } K, \sigma \models c > 0\right\} \\
&\overset{(\dagger)}{=} \bigcup_{\sigma \in \mathcal{S}} \bigcup_{K} \left\{\sigma' : (\sigma[\mathtt{i} \mapsto n], \sigma') \in \varrho(\mathtt{i--;}) \text{ and } n > 0\right\} \\
&= \bigcup_{\sigma \in \mathcal{S}} \left\{\sigma[\mathtt{i} \mapsto n],\ n \geq 0\right\}
\end{aligned}
$$

Here, step (†) follows by transferring the logic constraint $K, \sigma \models c > 0$ to the semantic level: Consider all substitutions of `i` in σ mapping to an arbitrary strictly positive number. The subsequent equality is correct because after subtracting 1 from a strictly positive number, the result will still be positive. Thus, $(s_2')^{\mathtt{i} \geq 0}$ is valid. ◊

3.2 SE Transition Relations, Exhaustiveness & Precision

In our framework, Symbolic Execution transitions connect symbolic *configurations*, which are sets of SESs. A configuration *Cnf* is related to a pair of states (I, O), where the input states $I \subseteq Cnf$ of the transition are mapped to the output states O, giving rise to a *successor configuration* resulting from replacing in *Cnf* the states I by the states O (formally, $Cnf \setminus I \cup O$). Traditionally, SE produces a Symbolic Execution *Tree*, i.e., one state in a configuration is replaced in each step by one ore more states in the successor configuration. Thus, I would always be a singleton set. SE transition relations with *state merging* [Kuz+12; SHB16] contain transitions from multiple input states to one output state, i.e., I contains more than one state and O is a singleton set. In Sect. 3.3, we discuss state merging techniques in more detail.

We define a more general notion of SE transition relation enabling transitions where neither I nor O are singletons: Not only 1-to-n transitions (as in traditional SE) and m-to-1 transitions (as usual for state merging) are allowed in our framework, but also general m-to-n transitions mapping any number of inputs to any number of outputs.

Definition 3.5 (SE Configuration and Transition Relation)**.** An *SE Configuration* is a set $Cnf \subseteq \mathbb{S}_{SE}$. An *SE Transition Relation* is a relation $\delta \subseteq 2^{\mathbb{S}_{SE}} \times (2^{\mathbb{S}_{SE}} \times 2^{\mathbb{S}_{SE}})$ associating to a configuration *Cnf* transitions $t = (I, O)$ of *input states* $I \subseteq Cnf$ and *output states* $O \subseteq 2^{\mathbb{S}_{SE}}$.[1]

[1] Note that in our understanding, the Cartesian product is *not* associative: Elements of δ are pairs where

We call $Cnf \setminus I \cup O$ the *successor configuration* of the transition t for Cnf. The relation δ is called SE Transition Relation *with (without) State Merging* if there is a (there is no) transition with more than one input state, i.e., $|I| > 1$. We write $Cnf \xrightarrow{t}_{\delta} Cnf'$ if $(Cnf, t) \in \delta$ and Cnf' is the successor configuration of t in Cnf. ◊

We defined our transition relations as general binary relations and not partial functions, which implies that there might be multiple possible successor configurations for an input configuration. The choice of which one to use falls to the employed SE *strategy*. This reflects reality: When symbolically executing a loop statement, a strategy might (for example based on a user-defined option) choose between a loop unwinding rule, a loop invariant rule or an abstract interpretation-based approach, to just name a few.

Most practical SE transition relations can be defined as a set of *schematic* SE *rules*, where each such rule represents a family of SE transitions (one transition for each consistent instantiation of the contained schematic placeholders). We denote SE rules either by describing the transition in tuple notation (I, O) or by using the notation of sequent calculi: Let $i_1, \ldots, i_m$, for $m > 0$, and $o_1, \ldots, o_n$, for $n \geq 0$, be SESs. The SE rule

$$\textsf{ruleName}\ \frac{o_1\ o_2\ \cdots\ o_n}{i_1\ i_2\ \cdots\ i_m}\ \text{(conditions)}$$

represents all instances of the SE transitions $(Cnf, (\{i_1, \ldots, i_m\}, \{o_1, \ldots, o_n\}))$ resulting from consistent replacement of schematic placeholders in the input and output states, for all suitable initial configurations Cnf s.t. $i_1, \ldots, i_m \in Cnf$. As for sequent calculus rules, the rule is read bottom-up, has a name (here "ruleName") written on the left and may have conditions written on the right. In the following, we show three example SE rules; further examples (of state merging rules) are given in Sect. 3.3.

Example 3.5 (SE Rules). Figure 3.1 shows three SE rules for assignment, conditional statement, and **`while`** loop, where for the latter, loop invariant-based abstraction is used. Schematic placeholders are C, $\mathcal{U}$, se, π, ω, etc. The schema variable $\mathcal{U}$ can be instantiated to any update, $\pi\omega$ to a Java context (as in the JavaDL calculus, see Sect. 2.2.4), se for a side effect-free expression, and so on. Let, for instance,

$$\begin{aligned} s &:= (\overbrace{\ \emptyset\ }^{C}, \overbrace{\mathtt{x} := \mathtt{z}}^{\mathcal{U}}, \overbrace{\mathbf{try}\ \{}^{\pi}\ \mathtt{y{=}x;}\ \overbrace{\mathtt{z{=}x;}\ \}\ \mathbf{finally}\ \{\}}^{\omega}) \\ s' &:= \big(\emptyset, (\mathtt{x} := \mathtt{z}) \circ (\mathtt{y} := \mathtt{x}), \mathbf{try}\ \{\ \mathtt{z{=}x;}\ \}\ \mathbf{finally}\ \{\}\big) \end{aligned}$$

the first component is an SES and the second again a pair of two SESs. In particular, δ is a *binary* relation, s.t. $dom(\delta) \subseteq 2^{\mathbb{S}_{SE}}$ and $rng(\delta) \subseteq 2^{\mathbb{S}_{SE}} \times 2^{\mathbb{S}_{SE}}$.

$$\text{assignment}\ \frac{(C, \mathcal{U} \circ \mathtt{x} := se, \pi\ \omega)}{(C, \mathcal{U}, \pi\ \mathtt{x=}se\mathtt{;}\ \omega)} \qquad \text{ifThenElse}\ \frac{\begin{array}{c}\big(C \cup \{\{\mathcal{U}\}(se \doteq \mathrm{TRUE})\}, \mathcal{U}, \pi\ p_1\ \omega\big) \\ \big(C \cup \{\{\mathcal{U}\}(se \doteq \mathrm{FALSE})\}, \mathcal{U}, \pi\ p_2\ \omega\big)\end{array}}{\big(C, \mathcal{U}, \pi\ \mathtt{\textbf{if}(}se\mathtt{)}\ \ p_1\ \ \mathtt{\textbf{else}}\ \ p_2\ \omega\big)}$$

$$\text{whileInv}\ \frac{\big(C \cup \{\{\mathcal{U} \circ \mathcal{U}_{havoc}\}(inv \land se \doteq \mathrm{FALSE})\}, \mathcal{U} \circ \mathcal{U}_{havoc}, \pi\ \omega\big)}{\big(C, \mathcal{U}, \pi\ \mathtt{\textbf{while}(}se\mathtt{)}\,body\ \omega\big)}\ (\star)$$

($\star$): The formula *inv* is a loop invariant for the loop in the program counter (i.e., initially valid and preserved at each further loop entry) and $\mathcal{U}_{havoc}$ an anonymizing update for *body*. The loop guard is assumed to be side effect-free, and both loop guard and body must not complete abruptly.

Figure 3.1: Some Example SE Rules

be two SESs and $s'' \in \mathbb{S}_{SE}$ an arbitrary SES. Then, the assignment rule covers the SE transition $(\{s, s''\}, (\{s\}, \{s'\}))$, i.e., replacement of s by s' in the configuration containing s and some s''. For the loop invariant rule, one first has to prove the non-trivial side condition that the formula *inv* is indeed a loop invariant. In a JavaDL sequent calculus rule, the corresponding two conditions would be proof branches on top of the rule. Actually, they are conditions on *inv* and not related to Symbolic Execution. One could remove the side conditions by using labeled SESs as follows:

$$\text{whileInv}\ \frac{\begin{array}{l}(C, \mathcal{U})^{inv} \\ \big(C \cup \{\{\mathcal{U} \circ \mathcal{U}_{havoc}\}(inv \land se \doteq \mathrm{TRUE})\}, \mathcal{U} \circ \mathcal{U}_{havoc}, body\big)^{inv} \\ \big(C \cup \{\{\mathcal{U} \circ \mathcal{U}_{havoc}\}(inv \land se \doteq \mathrm{FALSE})\}, \mathcal{U} \circ \mathcal{U}_{havoc}, \pi\ \omega\big)\end{array}}{\big(C, \mathcal{U}, \pi\ \mathtt{\textbf{while}(}se\mathtt{)}\,body\ \omega\big)}$$

As before, loop guard and body must not complete abruptly, and the loop guard is assumed to be side-effect free (Sect. 2.3 proposes an approach to handle abrupt completion, which we do not consider here for simplicity). This rule is an interesting use case for the integration of “pure” SE and validity reasoning with labeled SESs. ◊

We define big-step SE transition relations δ^* as the reflexive and transitive closure of SE transition relations δ.

Definition 3.6 (Big-Step SE Transition Relation)**.** Let δ be an SE transition relation. The *big-step SE transition relation δ^**, which is also an SE transition relation, is the reflexive and transitive closure of δ, i.e., if for $n \geq 0$, $Cnf_0 \xrightarrow{(I_1,O_1)}_{\delta} Cnf_1 \xrightarrow{(I_2,O_2)}_{\delta} \cdots \xrightarrow{(I_n,O_n)}_{\delta} Cnf_n$, then $Cnf_0 \xrightarrow{(Cnf_0 \setminus Cnf_n, Cnf_n \setminus Cnf_0)}_{\delta^*} Cnf_n$. $\Diamond$

Based on the concretization-based semantics of SESs, we define two aspects of the correctness of symbolic transition relations: *Exhaustiveness* and *precision*. Those properties are comparable to "recall" and "precision" in binary classification. Exhaustiveness is the property that during a symbolic transition, the set of concrete states represented by a configuration is not *decreased*, whereas precision is the property this set is not *increased*.

Definition 3.7 (Exhaustive SE Transition Relations)**.** An SE transition relation $\delta \subseteq 2^{\mathbb{S}_{SE}} \times (2^{\mathbb{S}_{SE}} \times 2^{\mathbb{S}_{SE}})$ is called *exhaustive* iff for each transition $(I,O) \in rng(\delta)$, $i \in I$ and concrete states $\sigma, \sigma' \in \mathscr{S}$, it holds that $\sigma' \in concr(i,\sigma)$ implies that there is an SES $o \in O$ and concrete state $\sigma'' \in \mathscr{S}$ s.t. $\sigma' \in concr(o,\sigma'')$. $\Diamond$

Definition 3.8 (Precise SE Transition Relations)**.** An SE transition relation $\delta \subseteq 2^{\mathbb{S}_{SE}} \times (2^{\mathbb{S}_{SE}} \times 2^{\mathbb{S}_{SE}})$ is called *precise* iff for each transition $(I,O) \in rng(\delta)$, $o \in O$ and concrete states $\sigma, \sigma' \in \mathscr{S}$, it holds that $\sigma' \in concr(o,\sigma)$ implies that there is an SES $i \in I$ and concrete state $\sigma'' \in \mathscr{S}$ s.t. $\sigma' \in concr(i,\sigma'')$. $\Diamond$

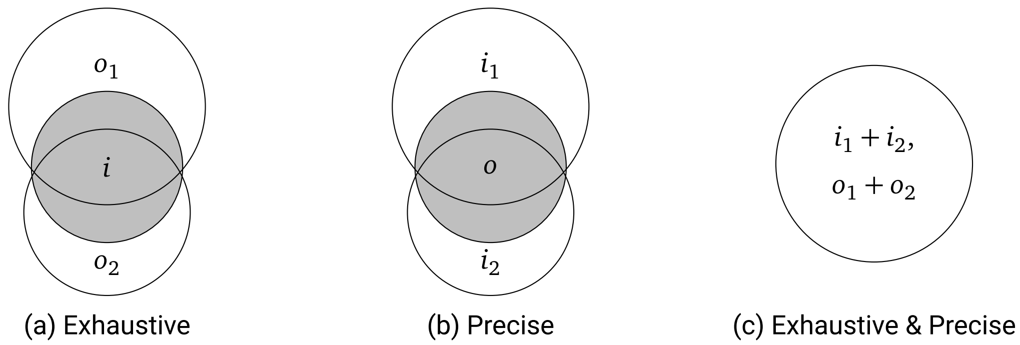


Figure 3.2: Visualization of Exhaustive and Precise SE Transitions

Figure 3.2 shows a visualization of Defs. 3.7 and 3.8 for a single SE step. Concretizations of input and output states are visualized as Venn diagrams. If two circles overlap, they have common concretizations. It is generally impossible to construct SE transition relations that are exhaustive *and* precise (Fig. 3.2c) for programs with loops and recursive calls, since those cannot always be precisely described in terms of First-Order Logic in the presence of symbolic bounds. In practice, this usually means that contracts are used to abstract from loops and recursive calls, allowing to process them sufficiently well.

For a rule-based SE transition relation, the exhaustiveness/precision of the whole relation follows from the exhaustiveness/precision of the individual SE rules. Defs. 3.7 and 3.8 directly transfer to rules, since they anyway quantify over all transitions.

The following lemma asserts that for exhaustive/precise SE transition relations δ, the corresponding big-step version δ^* is also exhaustive/precise.

Lemma 3.1 (Big-Step Exhaustiveness and Precision)**.** *Let δ be an exhaustive/precise SE transition relation. Then, the big-step SE transition relation δ^* is also exhaustive/precise.*

Proof. We show the exhaustiveness case, precision is analogous. Let δ be an exhaustive SE transition relation, δ^* be its big-step closure, $I, O \subseteq \mathbb{S}_{SE}$ be sets of SE states, $\mathit{Cnf}, \mathit{Cnf}'$ be SE configurations, and $\sigma, \sigma' \in \mathbb{S}_{SE}$ be concrete states s.t. $\mathit{Cnf} \xrightarrow{(I,O)}_{\delta^*} \mathit{Cnf}'$, and, for $i \in I$, $\sigma' \in \mathit{concr}(i, \sigma)$. To prove that δ^* is exhaustive, we have to show that there are $o \in O$, $\sigma'' \in \mathbb{S}_{SE}$ s.t. $\sigma' \in \mathit{concr}(o, \sigma'')$. Due to Def. 3.6, there is an $n \in \mathbb{N}$ s.t. $\mathit{Cnf} \xrightarrow{(I_1,O_1)}_{\delta} \mathit{Cnf}_1 \xrightarrow{(I_2,O_2)}_{\delta} \cdots \xrightarrow{(I_n,O_n)}_{\delta} \mathit{Cnf}'$. We prove by induction over n. The base cases $n = 0$ and $n = 1$ are either trivial (there is no i as above for $n = 0$) or otherwise follow from exhaustiveness of δ. For the induction step, the induction hypothesis (IH) is that $\mathit{Cnf} \xrightarrow{(I',O')}_{\delta^*} \mathit{Cnf}_n$ is an exhaustive transition, for $\mathit{Cnf} \xrightarrow{(I',O')}_{\delta^*} \mathit{Cnf}_n \xrightarrow{I_{n+1},O_{n+1}}_{\delta} \mathit{Cnf}'$. We distinguish the cases $i \in I'$ and $i \notin I'$.

Case $i \in I'$**:** Due to (IH), there are $o' \in O'$, σ''' s.t. $\sigma' \in \mathit{concr}(o', \sigma''')$. If $o' \notin I_{n+1}$, choose $o := o'$, $\sigma'' := \sigma'''$. If, however, $o' \in I_{n+1}$, due to exhaustiveness of δ there has to be a $o'' \in O_{n+1}$ s.t. there is a σ'''' with $\sigma' \in \mathit{concr}(o'', \sigma'''')$. Choose $o := o''$ and $\sigma'' := \sigma''''$.

Case $i \notin I'$**:** Then, the following holds for i:

$$
\begin{aligned}
i \in\ & \mathit{Cnf} \setminus \mathit{Cnf}' \setminus I' = \\
& \mathit{Cnf} \setminus (\mathit{Cnf}_n \setminus I_{n+1} \cup O_{n+1}) \setminus I' = \\
& \mathit{Cnf} \setminus ((\mathit{Cnf} \setminus I' \cup O') \setminus I_{n+1} \cup O_{n+1}) \setminus I' = \\
& \mathit{Cnf} \setminus I' \setminus ((\mathit{Cnf} \setminus I' \cup O') \setminus I_{n+1} \cup O_{n+1}) = \\
& (\mathit{Cnf} \setminus I') \cap I_{n+1} \setminus O_{n+1} = \\
& (\mathit{Cnf} \setminus I') \cap I_{n+1} \cap \overline{O_{n+1}}
\end{aligned}
$$

Therefore, it has to hold that $i \in I_{n+1}$. Then, there are $o' \in O_{n+1}$ and $\sigma''' \in \mathbb{S}_{SE}$ such that $\sigma' \in concr(o', \sigma''')$ due to exhaustiveness of δ. Since however, $o' \in O$ due to Def. 3.5, we can choose $o := o'$ and $\sigma'' := \sigma'''$. $\square$

For lightweight SE, *precise* transitions are desirable, since then, a bug found for an output state is feasible. In heavyweight SE, *exhaustiveness* is crucial, since anything proven for all output states of a step has to transfer to the inputs. To formalize this intuition, we define "strong" versions of precision and exhaustiveness, which prevent changing the interpretations of rigid symbols during transitions. Otherwise, we would have to restrict ourselves to postconditions without uninterpreted rigid function or predicate symbols.[2]

For strong precision, we simply fix a structure K for computing the concretizations, ensuring that all occurring rigid symbols are interpreted equally. In the case of strong exhaustiveness, we have to permit the addition of *fresh* symbols in a transition. This is, for instance, needed in loop invariant rules, where locations assigned in the loop body are anonymized by assigning them fresh constants. We therefore extend structures s.t. they can choose suitable instantiations for fresh constants, but have to preserve interpretations of symbols already present in the input states.

Definition 3.9 (Strongly Exhaustive SE Transition Relations). An SE transition relation $\delta \subseteq 2^{\mathbb{S}_{SE}} \times (2^{\mathbb{S}_{SE}} \times 2^{\mathbb{S}_{SE}})$ is called *strongly* exhaustive iff for each transition $(I, O) \in rng(\delta)$, $i \in I$, structure K and concrete states $\sigma, \sigma' \in \mathcal{S}$, it holds that $\sigma' \in concr_K(i, \sigma)$ implies that there is (1) a "conservative extension" K' of K interpreting all rigid symbols occurring in i the same way as K (in particular, $concr_K(i, \sigma) = concr_{K'}(i, \sigma)$), (2) an SES $o \in O$ and (3) a concrete state $\sigma'' \in \mathcal{S}$ s.t. $\sigma' \in concr_{K'}(o, \sigma'')$. $\Diamond$

Definition 3.10 (Strongly Precise SE Transition Relations). An SE transition relation $\delta \subseteq 2^{\mathbb{S}_{SE}} \times (2^{\mathbb{S}_{SE}} \times 2^{\mathbb{S}_{SE}})$ is called *strongly* precise iff for each transition $(I, O) \in rng(\delta)$, $o \in O$, structure K and concrete states $\sigma, \sigma' \in \mathcal{S}$, it holds that $\sigma' \in concr_K(o, \sigma)$ implies that there is an SES $i \in I$ and concrete state $\sigma'' \in \mathcal{S}$ s.t. $\sigma' \in concr_K(i, \sigma'')$. $\Diamond$

It is easy to see that the strong versions imply their weak counterparts. Lems. 3.2 and 3.3 use strong precision and strong exhaustiveness to formally express the intuitions about lightweight and heavyweight SE explained above.

[2] This would not bee too strict, since it holds for practically all postconditions. E.g., "pure" JML postconditions only contain program variables, logical variables bound by quantifiers, and *interpreted* functions like arithmetic operators, Java queries or elements of further predefined theories. The only way to include uninterpreted symbols in JML postconditions (in KeY) is to escape to JavaDL, e.g., using "**`\dl_`**".

Lemma 3.2 (A Bug Discovered by Strongly Precise SE Feasible). *Let δ be a strongly precise SE transition relation and $Cnf \xrightarrow{(I,O)}_{\delta*} Cnf'$. If a postcondition $\varphi \in \mathrm{Fml}$ is not true for a state $o \in Cnf'$, i.e., $\not\models o^{\varphi}$, it follows that there is an $i \in Cnf$ s.t. $\not\models i^{\varphi}$.*

Proof. The case $o \notin O$ is trivial (because then, $o \in Cnf$), therefore we assume $o \in O$. Since $\not\models o^{\varphi}$, there are K, σ s.t. $K, \sigma \not\models o^{\varphi}$, i.e. there is $\sigma' \in concr_K(o, \sigma)$ for which $K, \sigma' \not\models \varphi$. Because δ is strongly precise, there is a state $\sigma'' \in \mathcal{S}$ s.t. $\sigma' \in concr_K(i, \sigma'')$ for an $i \in I$ (if δ was only precise, and not *strongly* precise, we would obtain this for *another* K', and would have to additionally assume that φ has no uninterpreted rigid symbols). Therefore, it follows that $K, \sigma' \not\models \varphi$ and from this $K, \sigma'' \not\models i^{\varphi}$, i.e., $\not\models i^{\varphi}$. □

Lemma 3.3 (A Property Proven by Strongly Exhaustive SE Holds for the Inputs). *Let δ be a strongly exhaustive SE transition relation and $Cnf \xrightarrow{(I,O)}_{\delta*} Cnf'$. If a postcondition $\varphi \in \mathrm{Fml}$, which only contains rigid symbols already present in all states of Cnf, holds for all states $o \in Cnf'$, i.e., $\models o^{\varphi}$, it follows that for all $i \in Cnf$, it holds that $\models i^{\varphi}$.*

Proof. We assume that, for all $o \in Cnf'$, $\models o^{\varphi}$, and have to prove that, for all $i \in Cnf$, $\models i^{\varphi}$, i.e., $K, \sigma' \models \varphi$ for all K, σ and $\sigma' \in concr_K(i, \sigma)$. The case $i \notin I$ is trivial (because then, $i \in Cnf'$), therefore we assume $i \in I$. Since $\sigma' \in concr_K(i, \sigma)$ and δ is strongly exhaustive, there has to be a conservative w.r.t. i extension K' and concrete state σ'' s.t. for some output state $o \in O$, it holds that $\sigma' \in concr_{K'}(o, \sigma'')$. Because $\models o^{\varphi}$ (since $O \subseteq Cnf'$), we obtain $K', \sigma'' \models o^{\varphi}$ and therefore $K', \sigma' \models \varphi$. From this it follows that $K, \sigma' \models \varphi$ since φ only contains rigid symbols already occurring in Cnf and thus also in i, and K' is a conservative extension of K w.r.t. i. □

The subsequent Lem. 3.4 establishes a connection between the validity of a labeled SES and a standard JavaDL proof obligation for functional correctness of a program.

Lemma 3.4 (JavaDL and labeled SESs). *Let K be a structure, $s \in \mathcal{S}$ a concrete state, and Pre, Post $\in \mathrm{Fml}$ formulas. Then, it holds that $K, \sigma \models Pre \rightarrow \{\mathcal{U}\}[p]Post$ is equivalent to $K, \sigma \models (\{Pre\}, \mathcal{U}, p)^{Post}$.*

Proof. Let $s := (\{Pre\}, \mathcal{U}, p)$. The right-hand side of the equivalence to show, $K, \sigma \models s^{Post}$, is equivalent to $K, \sigma' \models Post$ for $\sigma' \in concr_K(s, \sigma)$. This in turn is equivalent to (1) either $K, \sigma \not\models Pre$ or p does not terminate in $val(K, \sigma|\mathcal{U})(\sigma)$ (because then, there is no σ'), or (2) σ' is such that $(val(K, \sigma|\mathcal{U})(\sigma), \sigma') \in \varrho(p)$. This, however, is exactly the meaning of the left-hand side of the equivalence: Either K and σ do not satisfy Pre, or p does not terminate in the state resulting from the update $\mathcal{U}$, or in the resulting state after termination of p, the postcondition *Post* holds. $\square$

By Lem. 3.4, any labeled SES can be easily transformed to a logically equivalent JavaDL formula. However, it is not always possible to transform a JavaDL formula to a labeled SES. For instance, the formula might contain multiple modalities (then, it is not clear which program should be the program counter), or have an unsuitable structure.

The main idea behind heavyweight SE is to prove a property of a program by reducing the problem to a number of first-order assertions by stepwise reduction of the program counter to branch conditions and changes to the symbolic store. Those first-order problems can then ideally be discarded automatically by a theorem prover or Satisfiability Modulo Theories (SMT) solver. The following corollary is implied by Lems. 3.3 and 3.4.

Corollary 3.5 (Main Principle of Heavyweight SE)**.** *If, for a strongly exhaustive SE transition relation δ, $\{(\{Pre\}, \mathcal{U}, p)\} \xrightarrow{t}_{\delta^*} Cnf$ s.t. the final configuration Cnf consists of a set of output states with empty program counters $(C_i, \mathcal{U}_i)$ and all first order proof obligations $C_i \rightarrow \{\mathcal{U}_i\}Post$ hold for a postcondition Post (which only contains rigid symbols already present in the input state), then the JavaDL formula $Pre \rightarrow \{\mathcal{U}\}[p]Post$ is valid.*

Cor. 3.5 relates (strong) *exhaustiveness* to *soundness* of an SE-based validity calculus, e.g., JavaDL (see Prop. 2.1). If we can derive a sequent using strongly exhaustive SE transitions and sound first-order inference steps, then it is valid. Also *completeness* (Prop. 2.2) and (strong) *precision* are related: We generally have to require precise transitions for completeness, because too strong abstraction, e.g., by loop invariants, can cause that valid assertions cannot be derived.

In the next section, we illustrate these concepts along concrete SE rules for *state merging*. Some of those are precise and exhaustive, and some only satisfy one of these properties.

3.3 State Merging

One of the main bottlenecks of Symbolic Execution is the *path explosion problem* [CS13; Bal+18; Yan+19]. It stems from the fact that SE must explore all symbolic paths of a program to achieve high coverage (in testing), respectively, soundness (in verification). As a consequence, the number of paths from the root to the leaves in a Symbolic Execution Tree is usually *exponential* in the number of static branches of the executed program.

Various strategies are in use to mitigate path explosion, from unsound (i.e., underapproximating) techniques like preconditioned [Avg+14] and under-constrained SE [ED07] via subsumption techniques [APV06; CJM14; JMN13; Jaf+12] to sound (i.e., overapproximating) approaches like method contracts [Ahr+16] and value summaries [Sen+15]. The last two allow performing SE per method: Different symbolic execution paths are merged into the postcondition of a contract or a value summary (a conditional execution state over guard expressions). Summaries are computed on the fly and bottom-up, while contracts characterize all possible behaviors and must at least partially be written by hand. Unfortunately, even the use of rich contracts (instead of inlining) is not sufficient to deal with state explosion in complex problems [Gou+15; Gou+19].

A common technique to alleviate state explosion in SETs consists of *merging* the states resulting from an SE step that caused a split (e.g., guard evaluation, statements that can throw exceptions, polymorphic method calls). Indeed, several state merging variants were suggested for SE [HSS09; Kuz+12; Sen+15]. Similar to accounts on SE in general, existing literature lacks a unifying view on state merging itself. Instead, they focus on a specific state merging technique (e.g., the above mentioned value summaries) or, e.g., efficient application of state merging [Kuz+12]. In particular, the question of when a state merging technique is *sound*, i.e., can be used for program proving, is left open.

In [Sch15; SHB16] we presented a "general lattice-based framework for merging symbolic execution states". This framework formulates conditions to state merging operations and derives a correctness result for a state merging SE rule only using operations satisfying these conditions. The imposed five conditions for correct merge operations consist of three basic lattice properties (idempotency, commutativity, and associativity), and two properties for correctness. Of the latter, one requires that the concretizations of the input states are also present in the output state. The last property requires that logical axioms about Skolem symbols introduced by a merge operation are satisfiable.

Our SE theory of Sect. 3.2 is strictly more general than the lattice-based state merging framework: Exhaustiveness of m-to-n SE transition relations subsumes *both* properties related to correctness of merge operations. Furthermore, the theory also facilitates discussing the *precision* of merge operations (that not necessarily need to be exhaustive, if not used for program proving). The lattice properties, on the other hand, are not needed for useful merge operations, neither in the context of lightweight nor heavyweight SE, which is why we decide to not impose them on merge operations.

In the remainder of this section, we introduce two practically relevant state merging operations also discussed in [SHB16], *If-Then-Else Merging* and *Predicate Abstraction*, and argue that they are strongly precise and exhaustive in the case of the first, and strongly exhaustive only in the case of the second operation. We furthermore regard two variants of the If-Then-Else merge technique based on case distinctions in the merged path condition. The first is precise and exhaustive, the second only exhaustive. To showcase a precise, but *not* exhaustive merge technique, we define an operation *dropping* one SE branch. This corresponds to path pruning, as performed by many lightweight SE approaches. We define all techniques on *two* input states *with the same program counters*. The latter restriction is crucial, since there is no "merge" operator for Java programs (like disjunction in logic). The former simplifies the presentation, but is not necessary. The lattice-based framework requires associative and commutative merge operations, making it is easy to justify the restriction to two inputs. Also this is superfluous: Exhaustiveness/precision is always guaranteed, whether we merge more than two states in one or in more steps.

We first define two normal forms for updates.

Definition 3.11 (Parallel Normal Form of Updates)**.** An update $\mathcal{U} \in \mathrm{Upd}$ is in *Parallel Normal Form (PNF)* if it is either the empty update *Skip* or has the structure $\mathcal{U}_1 \,||\, \mathcal{U}_2 \,||\, \ldots \,||\, \mathcal{U}_n$ for $n \geq 1$, where each $\mathcal{U}_i$ is an elementary update. It is in *Conflict-Free Parallel Normal Form (CF-PNF)* if additionally, the update is conflict-free, i.e., no two elementary updates have the same left-hand side. ◊

Note that each update can be converted to an update in PNF/CF-PNF by transforming sequential updates as in $\mathcal{U}_1 \circ \mathcal{U}_2$ or $\{\mathcal{U}_1\}\{\mathcal{U}_2\}\varphi$ to parallel ones $\mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2$ and by resolving conflicts through removal of earlier occurrences of updates to conflicting program variables ("update parallelization", see also [Ahr+16, Sect. 15.2]).

The If-Then-Else technique and related merge techniques require that path conditions are "separable", meaning, they cannot be true at the same time.[3] We formally define separable formulas (and SESs) as follows:

Definition 3.12 (Separable Formulas and SESs)**.** We say that two formulas $\varphi_1, \varphi_2 \in \mathrm{Fml}$ are *separable* / can be *separated* iff it holds that $\models \neg(\varphi_1 \wedge \varphi_2)$. Two SESs $(\mathcal{U}_i, C_i, p_i)$, $i = 1, 2$, are separable if $\bigwedge C_1$ and $\bigwedge C_2$ are separable. ◊

For instance, the formulas $\varphi_1 \wedge \mathtt{x} > 0$ and $\varphi_2 \wedge \mathtt{x} \leq 0$ are separable, for any program variable $\mathtt{x}$ and $\varphi_1, \varphi_2 \in \mathrm{Fml}$, since $\mathtt{x} > 0$ and $\mathtt{x} \leq 0$ can never be true simultaneously. We call the formula $\mathtt{x} > 0$ a *separating formula*, since it is implied by the first of the separable

[3] Note that by this definition, all *unsatisfiable* formulas are separable, because they are never at the same time true. E.g., "false" is separable from itself. This could be prevented by demanding satisfiability of the formulas, which however is practically complicated and not necessary for state merging.

formulas, and its *negation* is implied by the other. The existence of separating formulas follows from Craig-Lyndon interpolation [Cra57b; Cra57a; Lyn59]. We show a simplified version of Lyndon's theorem in the variant of [TS96]. In the following definition, a *theory* for a set of axioms is a set of closed JavaDL formulas obtained by recursive application of JavaDL inference rules on the axioms. We refer to [TS96] for a proof of the theorem.

Theorem 3.6 (Interpolation (Lyndon)). *Let T_1, T_2 be two theories such that their union $T_1 \cup T_2$ is unsatisfiable. Then, there is an* interpolant ψ *in the intersection of the languages of T_1 and T_2 which is true in all models of T_1, and false in all models of the T_2. Moreover, every relation symbol which occurs positively in ψ occurs positively in some formula of T_1 and negatively in some formula of T_2; conversely, relation symbols with negative occurrence in ψ occur negatively in some formula of T_1 and positively in some formula of T_2.*

Corollary 3.7 (Existence of Separating Formulas). *Two formulas $\varphi_1, \varphi_2 \in \mathrm{Fml}$ are separable if, and only if, there exists a* separating formula $\psi \in \mathrm{Fml}$ *s.t.* $\models \varphi_1 \rightarrow \psi$ *and* $\models \varphi_2 \rightarrow \neg\psi$.

Proof. Let φ_1, φ_2 be two separable formulas and, for $i = 1, 2$, T_i be the smallest theory including φ_i. Since $\models \neg(\varphi_1 \land \varphi_2)$, the theory $T_1 \cup T_2$ is unsatisfiable. From Thm. 3.6, we obtain an interpolant ψ s.t. $T_1 \models \psi$ and $T_2 \models \neg\psi$. Since T_1 only consists of consequences of φ_1, it follows that $\varphi_1 \models \psi$, i.e., $\models \varphi_1 \rightarrow \psi$. Similarly, it follows that $\models \varphi_2 \rightarrow \neg\psi$. Therefore, ψ is a separating formula, which additionally satisfies the constraints on positive and negative occurrences of relation symbols.

The reverse direction is not implied by Lyndon's theorem, but not difficult: Assume that ψ is a separating formula, and $\models \neg(\varphi_1 \land \varphi_2)$ does *not* hold, i.e., $K, \sigma \models \varphi_1 \land \varphi_2$ for some structure K and state σ. Then, however, it follows that $K, \sigma \models \varphi_1$ and $K, \sigma \models \varphi_2$, and, since ψ is a separating formula, $K, \sigma \models \psi$ and $K, \sigma \models \neg\psi$, which is a contradiction. Therefore, φ_1 and φ_2 are separable. $\square$

In the example from above, the separating formula $\mathtt{x} > 0$ for $\varphi_1 \land \mathtt{x} > 0$ and $\varphi_2 \land \mathtt{x} \leq 0$ is also a Craig-Lyndon interpolant: The relation ">" occurs positively in the theory for the first, and negatively in the theory for the second formula (since $\mathtt{x} \leq 0 \equiv \neg \mathtt{x} > 0$).

States suitable for state merging are usually separable, since SE normally branches to perform *case distinctions*. An example is SE of an **`if`** statement: In one branch, the guard of the statement is assumed to hold, and in the other its negation. Consequently, SESs for these branches are separable, and the guard of the conditional statement is a separating formula for those path conditions.

We subsequently define the If-Then-Else state merging technique.

Definition 3.13 (The If-Then-Else Merge Rule)**.** Let, for $i = 1, 2$, $s_i = (C_i, \mathcal{U}_i, p)$ be two SESs where ($\star$) the $\mathcal{U}_i$ are in CF-PNF, and $\mathbb{C}_i := \bigwedge C_i$. We require (†) that s_1 and s_2 are separable. Then, the SE rule $\mathsf{ifThenElseMerge}$ is defined as

$$\mathsf{ifThenElseMerge}\ \frac{\left(\{\mathbb{C}_1 \vee \mathbb{C}_2\}, \mathsf{x}_1 := t_{\mathsf{x}_1} \,||\, \mathsf{x}_2 := t_{\mathsf{x}_2} \,||\, \ldots \,||\, \mathsf{x}_n := t_{\mathsf{x}_n}, p\right)}{(\mathcal{U}_1, C_1, p) \quad (\mathcal{U}_2, C_2, p)}\ (\star), (\dagger)$$

where (1) the $\mathsf{x}_1, \ldots, \mathsf{x}_n$ are the left-hand sides of the updates $\mathcal{U}_1$, $\mathcal{U}_2$, such that each x_i occurring in either $\mathcal{U}_1$ or $\mathcal{U}_2$ (or both) occurs exactly once in the output state, (2) t_{x_i} is term $\mathit{if}\,(\varphi)\,\mathit{then}\,(t^1_{\mathsf{x}_i})\,\mathit{else}\,(t^2_{\mathsf{x}_i})$ where φ is a separating formula for s_1 and s_2, and $t^j_{\mathsf{x}_i}$ is the right-hand side of x_i in $\mathcal{U}_j$ or, if x_i is not contained in $\mathcal{U}_j$, the variable x_i itself. ◊

Separability is required for the If-Then-Else merge rule since in cases with "overlapping" path conditions, the conditionals used as right-hand sides in the symbolic store of the merged state might evaluate wrongly in a given concrete state.[4] Lems. 3.8 and 3.9 assert that the merging SE rule $\mathsf{ifThenElseMerge}$ is strongly exhaustive and precise.

Lemma 3.8. *The SE rule* $\mathsf{ifThenElseMerge}$ *is strongly exhaustive.*

Proof. Let $s' = \left(C', \mathcal{U}', p\right)$ be the output state of an application of $\mathsf{ifThenElseMerge}$. We have to show that for states $\sigma, \sigma' \in \mathcal{S}$, structure K, and $i = 1, 2$, $\sigma' \in \mathit{concr}_K(s_i, \sigma)$ implies that there is a $\sigma'' \in \mathcal{S}$ and conservative extension K' s.t. $\sigma' \in \mathit{concr}_{K'}(s', \sigma'')$. We show that this is the case for $\sigma'' := \sigma$ and $K' := K$. Since $\sigma' \in \mathit{concr}_K(s_i, \sigma)$, it holds that $K, \sigma \models \mathbb{C}_i$ and, where $\sigma''' = \mathit{val}(K, \sigma | \mathcal{U}_i)(\sigma)$, $(\sigma''', \sigma') \in \varrho(p)$. Therefore, the path condition C' of s' is also satisfied: $K, \sigma \models \mathbb{C}_1 \vee \mathbb{C}_2$. Additionally, we also have $\mathit{val}\left(K, \sigma | \mathcal{U}'\right)(\sigma) = \sigma'''$: Since φ is a separating formula, the guard of the conditional formula which is the right-hand side for a variable x_k evaluates in σ exactly to the right-hand side in $\mathcal{U}_i$ or, if x_k is not present in $\mathcal{U}_i$, to the variable x_k, and does therefore not change the valuation of the update. It follows that $\sigma' \in \mathit{concr}_K(s', \sigma)$. □

Lemma 3.9. *The SE rule* $\mathsf{ifThenElseMerge}$ *is strongly precise.*

Proof. The argument is similar to the one of Lem. 3.8. Since C_1 and C_2 are separable, any concrete state σ can only satisfy one of C_1 or C_2. Therefore, the output state s' "collapses"

[4] The If-Then-Else rule of [SHB16] is, strictly speaking, unsound / not exhaustive, since it does not require the input states to be separable, and only mentions separating formulas as a possible simplification.

to one of the input states s_1 or s_2. This one is then chosen to reach the conclusion that ifThenElseMerge is precise. $\square$

A variant of ifThenElseMerge, pathCondMerge, performs case distinctions not by conditional terms, but by implications in the path condition of the merged state. Both rules are strongly exhaustive and precise, and have the same side conditions, i.e., are logically equivalent. One can, for instance, use pathCondMerge instead of ifThenElseMerge if conditional terms are not available in the underlying logic. This choice has an impact on proof performance: Depending on the particular problem, any of the two merge rules can perform better or worse than the other [Sch15], since conditional terms and implications in path conditions are most likely differently treated by proof strategies.

Definition 3.14 (The Path Condition Merge Rule)**.** Let, for $i = 1, 2$, $s_i = (C_i, \mathcal{U}_i, p)$ be two SESs where ($\star$) the $\mathcal{U}_i$ are in CF-PNF, and $\mathbb{C}_i := \bigwedge C_i$. We require (†) that s_1 and s_2 are separable. Then, the SE rule pathCondMerge is defined as

$$\text{pathCondMerge}\quad \frac{\Big(\{\mathbb{C}_1 \lor \mathbb{C}_2\} \cup \bigcup_{j=1}^{2} \Big\{\varphi_j \rightarrow \big(\bigwedge_{i=1}^{n} c_{\mathsf{x}_i} \doteq t^j_{\mathsf{x}_i}\big)\Big\}, \mathsf{x}_1 := c_{\mathsf{x}_1} \,\|\, \dots \,\|\, \mathsf{x}_n := c_{\mathsf{x}_n}, p\Big)}{(C_1, \mathcal{U}_1, p) \quad (C_2, \mathcal{U}_2, p)}\ (\star), (\dagger)$$

where (1) ψ is a separating formula for $\mathbb{C}_1$ and $\mathbb{C}_2$, φ_1 is ψ and φ_2 is $\neg\psi$, (2) the $\mathsf{x}_1, \dots, \mathsf{x}_n$ are the left-hand sides of the updates $\mathcal{U}_1$, $\mathcal{U}_2$, such that each x_i occurring in either $\mathcal{U}_1$ or $\mathcal{U}_2$ (or both) occurs exactly once in the output state, (3) the c_{x_k} are fresh Skolem constants of suitable types, (4) $t^j_{\mathsf{x}_i}$ is the right-hand side of x_i in $\mathcal{U}_j$ or, if x_i is not contained in $\mathcal{U}_j$, the variable x_i itself. $\Diamond$

We omit proving the exhaustiveness / precision lemma for pathCondMerge; it is similar to the proofs of Lems. 3.8 and 3.9. The difference is that for strong exhaustiveness, a conservative extension of the structure has to be chosen which interprets fresh constants according to the right-hand sides of the symbolic store of the current input state.

Lemma 3.10. *The SE rule* pathCondMerge *is strongly exhaustive and precise.*

From rule pathCondMerge, we can create a derivative which is not precise, but does *not require separability* of the input states, by changing the constraints on the new Skolem constants in the path condition from two "guarded conjunctions" to simple disjunctions. The resulting rule disjunctionMerge is defined as follows:

Definition 3.15 (The Disjunction Merge Rule). Let, for $i = 1, 2$, $s_i = (\mathcal{U}_i, C_i, p)$ be two SESs where $(\star)$ the $\mathcal{U}_i$ are in CF-PNF, and $\mathbb{C}_i := \bigwedge C_i$. Then, the SE rule $\mathsf{disjunctionMerge}$ is defined as

$$\mathsf{disjunctionMerge}\quad \frac{\left(\{\mathbb{C}_1 \vee \mathbb{C}_2\} \cup \bigcup_{i=1}^{n} \left\{ \bigvee_{j=1}^{2} \left(c_{\mathsf{x}_i} \doteq t^j_{\mathsf{x}_i}\right)\right\}, \mathsf{x}_1 := c_{\mathsf{x}_1} \,\|\, \ldots \,\|\, \mathsf{x}_n := c_{\mathsf{x}_n}, p\right)}{(C_1, \mathcal{U}_1, p) \quad (C_2, \mathcal{U}_2, p)} \;(\star)$$

where (1) the $\mathsf{x}_1, \ldots, \mathsf{x}_n$ are the left-hand sides of the updates $\mathcal{U}_1$, $\mathcal{U}_2$, such that each x_i occurring in either $\mathcal{U}_1$ or $\mathcal{U}_2$ (or both) occurs exactly once in the output state, (2) the c_{x_k} are fresh Skolem constants of suitable types, (3) $t^j_{\mathsf{x}_i}$ is the right-hand side of x_i in $\mathcal{U}_j$ or, if x_i is not contained in $\mathcal{U}_j$, the variable x_i itself. $\Diamond$

Lemma 3.11. *The SE rule* $\mathsf{disjunctionMerge}$ *is strongly exhaustive.*

Proof. Let $s' = \left(C', \mathcal{U}', p\right)$ be the output state of an application of $\mathsf{disjunctionMerge}$. We have to show that, for a structure K with interpretation function I, concrete states $\sigma, \sigma' \in \mathcal{S}$ and $j = 1, 2$, $\sigma' \in concr_K(s_j, \sigma)$ implies that there is a conservative extension K' of K with interpretation function I' and $\sigma'' \in \mathcal{S}$ s.t. $\sigma' \in concr_{K'}(s', \sigma'')$. We show that this is the case for $\sigma'' := \sigma$ and K' defined as K apart from the interpretation of the new Skolem constants c_{x_i}, for which we set $I'(c_{\mathsf{x}_i}) := val\left(K, \sigma | t^j_{\mathsf{x}_i}\right)$. Observe that K' is a conservative extension since the constants do not occur in the input state. Since $\sigma' \in concr_K(s_j, \sigma)$, it holds that $K, \sigma \models C_j$, and, where $\sigma''' = val\left(K, \sigma | \mathcal{U}_j\right)(\sigma)$, $(\sigma''', \sigma') \in \varrho(p)$. Because the Skolem constants are introduced freshly, they do not occur in C_j and we also have $K', \sigma \models \bigwedge C_j$; the chosen valuation of the Skolem constants in I' ensures that the second part of C' consisting of the disjunctive equations is also satisfied. Furthermore, we also have $val\left(K', \sigma | \mathcal{U}'\right)(\sigma) = \sigma'''$, since the Skolem constants evaluate in K' exactly to the values of the right-hand sides in $\mathcal{U}_j$ or, if a x_i is not present in $\mathcal{U}_j$, to the value of the variable itself. It follows that $\sigma' \in concr_{K'}(s', \sigma)$. $\square$

The following example demonstrates an application of state merging and also that $\mathsf{disjunctionMerge}$ is not precise.

Example 3.6 (State Merging with $\mathsf{disjunctionMerge}$). Let

$$\begin{aligned} s_1 &:= (\{\mathsf{y} \geq 0\},\ \mathsf{x} := 1 \,\|\, \mathsf{y} := 2,\ \mathsf{z{=}3\,;}\,) \\ s_2 &:= (\{\mathsf{y} < 0\},\ \mathsf{x} := 4,\ \mathsf{z{=}3\,;}\,) \end{aligned}$$

We can merge these SESs using the disjunctionMerge rule, since they have the same program counters and the symbolic stores are in CF-PNF. The resulting SES is

$$s' = \big(\{\mathsf{y} \geq 0 \vee \mathsf{y} < 0,\, c_{\mathsf{x}} \doteq 1 \vee c_{\mathsf{x}} \doteq 4,\, c_{\mathsf{y}} \doteq 2 \vee c_{\mathsf{y}} \doteq \mathsf{y}\},\ \mathsf{x} := c_{\mathsf{x}} \,\|\, \mathsf{y} := c_{\mathsf{y}},\ \mathtt{z=3;}\big)$$

It is straightforward to see that each concretization of the input states is also represented by the output state; one just has to "pick" the right disjunct in the path condition. However, the set of concretizations for s' contains a concrete state mapping x to 4 and y to 2; this concrete state is contained neither in the set of concretizations of s_1 nor of s_2. Thus, disjunctionMerge is imprecise. We chose an example with separable input states, since this is what usually is found in practice; nevertheless, this is not required by disjunctionMerge, in contrast to ifThenElseMerge and pathCondMerge. ◊

Predicate abstraction [GS97; FQ02] is an abstract interpretation [CC77] technique in which the abstract domain is constructed from a set of unary predicates. This is particularly interesting in applications where the abstract domain should be constructed ad-hoc from a postcondition, for example in loop invariant generation—or state merging.

Our framework for predicate abstraction is based on *predicate abstraction structures* consisting of (1) a carrier set of predicates, (2) a join operation, and (3) an abstraction function. Predicates map terms to formulas. For instance, a predicate representing strictly positive numbers can be defined as $\lambda t.t > 0$. Join operations are standard. They take two predicates and output their lowest upper bound. If, in the induced lattice, a predicate p_1 is smaller (more specific) than a predicate p_2, it has to hold that $p_1(t) \rightarrow p_2(t)$, for all terms t of suitable type. For instance, $\lambda t.t > 0$ is smaller than $\lambda t.t \geq 0$, and indeed, it holds that $t > 0 \rightarrow t \geq 0$ for any term t. Abstraction functions α take a path condition and a term and output a predicate which soundly abstracts the term. The path condition is needed for precision: If the term is symbolic, one can otherwise only provide a very coarse abstraction. For instance, the term c, for an uninterpreted numeric constant c, can represent any number; under the path condition $c > 0$, however, it represents strictly positive numbers only. An abstraction for a term t under path condition C is sound if the returned predicate, when applied to t, is valid under C. Consider again the term c and path condition $c > 0$. A sound abstraction is, for example, the predicate $\lambda t.t > 0$. Applying this to c leads to $c > 0$, which is trivially implied by C. The predicate $\lambda t.t < 0$ representing strictly negative numbers is, e.g., not a sound abstraction for c under C.

We formally define predicate abstraction structures.

Definition 3.16 (Predicate Abstraction Structure)**.** A *predicate abstraction structure* is a tuple $(\mathit{Preds}, \sqcup, \alpha)_A$ where A is a type, *Preds* is a set of unary predicates $p(t)$ mapping terms of type A to closed formulas (formally, the p are functions $\mathrm{Trm}_A \to \mathrm{Fml}$), $\sqcup : \mathit{Preds} \times \mathit{Preds} \to \mathit{Preds}$ is a total operation on *Preds*, and $\alpha : 2^{\mathrm{Fml}} \times \mathrm{Trm}_A \to \mathit{Preds}$ is a total function from sets of formulas (i.e., path conditions) and terms of type A to *Preds*. We define the equality relation $\doteq$ and the *induced partial order relation* $\preceq$ of $\sqcup$ as

$$\begin{array}{rcl} \doteq & := & \{(p_1, p_2) : \models \forall v : A, (p_1(v) \leftrightarrow p_2(v))\} \\ \preceq & := & \{(p_1, p_2) : (p_1 \sqcup p_2) \doteq p_2\} \end{array}$$

We impose the following conditions:

(1) *Lattice:* $(\mathit{Preds}, \sqcup)$ is a join-semilattice, i.e., $\sqcup$ is associative, commutative and idempotent for the equality relation $\doteq$, and $(\mathit{Preds}, \preceq)$ is a partially ordered set,

(2) *Sound Abstraction:* for all $C \subseteq \mathrm{Fml}$ and $t \in \mathrm{Trm}_A$, it holds that $C \models \alpha(C, t)(t)$,

(3) *Logic Order:* for all $t \in \mathrm{Trm}_A$ and $p_1, p_2 \in \mathit{Preds}$, $p_1 \preceq p_2$ implies $\models p_1(t) \to p_2(t)$. $\Diamond$

The following example introduces a predicate abstraction structure for sign analysis.

Example 3.7 (Sign Analysis Predicate Abstraction Structure)**.** The sign analysis predicate abstraction structure $sgn = (\mathit{Preds}_{sgn}, \sqcup_{sgn}, \alpha_{sgn})_{int}$ is defined as follows: First, set

$$\mathit{Preds}^0_{sgn} := \{\lambda t.t < 0, \lambda t.t > 0, \lambda t.t \doteq 0\}$$

The carrier set Preds_{sgn} is then the set of all disjunctions of Preds^0_{sgn}:

$$\mathit{Preds}_{sgn} := \left\{\lambda t. \bigvee \{p_1(t), \ldots, p_n(t)\} : \{p_1, \ldots, p_n\} \subseteq \mathit{Preds}^0,\ n \geq 0\right\},$$

or concretely the set

$$\begin{array}{rl} \mathit{Preds}_{sgn} = \{ & \lambda t.t < 0, \lambda t.t > 0, \lambda t.t \doteq 0, \\ & \lambda t.\,\mathrm{false}, \lambda t.(t < 0 \lor t \doteq 0), \lambda t.(t > 0 \lor t \doteq 0), \\ & \lambda t.(t < 0 \lor t > 0), \lambda t.(t < 0 \lor t \doteq 0 \lor t > 0)\} \end{array}$$

Figure 3.3 represents the join operator $\sqcup_{sgn}$ and partial order $\preceq_{sgn}$ of sgn: If two elements p_1, p_2 in the depicted lattice are connected by a line (and p_2 is not on a lower level than p_1), it holds that $p_1 \preceq_{sgn} p_2$. The whole relation $\preceq_{sgn}$ is the reflexive and transitive closure of this. To compute the join $p_1 \sqcup_{sgn} p_2$ of two elements, compute the supremum of p_1 and p_2 in $(\mathit{Preds}_{sgn}, \preceq_{sgn})$. The α_{sgn} function is defined such that $\alpha_{sgn}(C, t)$ is the least element p of Preds_{sgn} for which the formula $\bigwedge C \to p(t)$ can be proven. Note that this always holds for the top element of the lattice since the bound expression is logically equivalent to true; if

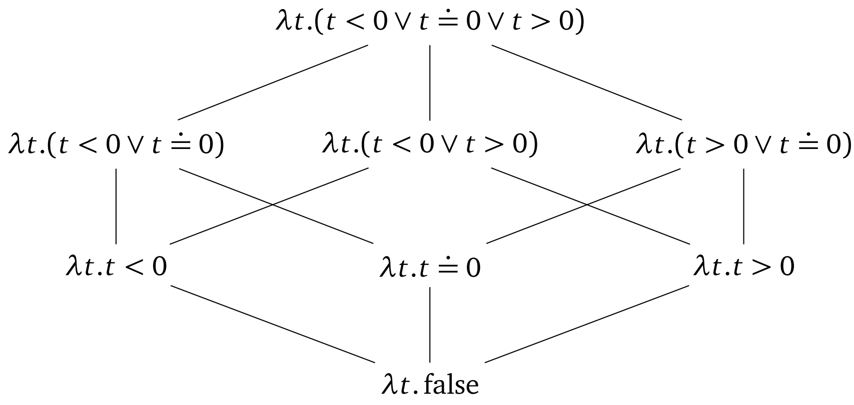


Figure 3.3: Sign Analysis Predicate Abstraction Lattice

this was not the case, we would have to add such a "default" element such that it is always possible to compute the abstraction of a term and the join of two predicates. For example, let $C := \{\mathsf{x} > \mathsf{y}, \mathsf{y} = -1\}$. Then, $\alpha_{sgn}(C, \mathsf{x}) = \lambda t.t > 0 \vee t \doteq 0$, since $\bigwedge C \rightarrow (\mathsf{x} > 0 \vee \mathsf{x} \doteq 0)$ is provable and there is no suitable lower element in the lattice. The abstract representation of variable y for the same path condition is $\alpha_{sgn}(C, \mathsf{y}) = \lambda t.t < 0$. If we construct the join of these predicates, we end up with the top predicate since it is the supremum of those in $(Preds_{sgn}, \preceq_{sgn})$. ◊

The lattice in Example 3.7 is constructed from all disjunctive combinations of a set of "core" predicates. This is a common pattern: If the core predicates cover the complete state space, no explicit top predicate has to be added. Alternatively, one can also construct the domain from all *conjunctions* (see [SHB16] for an example), or use the core predicates only, in which case one has to ensure the existence of a top element. Based on predicate abstraction structures, we can create a merge rule which is structurally similar to pathCondMerge, but uses abstracted instead of concrete values in the merged state. The rule uses a whole family of such structures s.t. there is one for each occurring type. In practice, one can fall back to, e.g., disjunctive merging if there is no available predicate abstraction structure for a particular type.

Definition 3.17 (The Predicate Abstraction Merge Rule)**.** Let, for $i = 1, 2$, $s_i = (C_i, \mathcal{U}_i, p)$ be two SESs where ($\star$) the $\mathcal{U}_i$ are in CF-PNF, and $\mathbb{C}_i := \bigwedge C_i$. Let furthermore (†) $\mathbb{P} = \{(Preds, \sqcup, \alpha)_A\}_A$ be a family of predicate abstraction structures for each type $A \in \mathrm{TSym}$. Then, the SE rule predicateAbstrMerge is defined as predicateAbstrMerge

$$\frac{\left(\{\mathbb{C}_1 \vee \mathbb{C}_2\} \cup \bigcup_{i=1}^{n} \left\{(\alpha(C_1, t^1_{\mathsf{x}_i}) \sqcup \alpha(C_2, t^2_{\mathsf{x}_i}))(c_{\mathsf{x}_i})\right\}, \mathsf{x}_1 := c_{\mathsf{x}_1} \parallel \ldots \parallel \mathsf{x}_n := c_{\mathsf{x}_n}, p\right)}{(C_1, \mathcal{U}_1, p) \quad (C_2, \mathcal{U}_2, p)} \; (\star), (\dagger)$$

where (1) the $\mathsf{x}_1, \ldots, \mathsf{x}_n$ are the left-hand sides of the updates $\mathcal{U}_1$, $\mathcal{U}_2$, such that each x_i occurring in either $\mathcal{U}_1$ or $\mathcal{U}_2$ (or both) occurs exactly once in the output state, (2) the c_{x_k} are fresh Skolem constants of suitable types, (3) $t^j_{\mathsf{x}_i}$ is the right-hand side of x_i in $\mathcal{U}_j$ or, if x_i is not contained in $\mathcal{U}_j$, the variable x_i itself, and (4) we write α, $\sqcup$ for the appropriate functions of the structure $(\mathit{Preds}, \sqcup, \alpha)_A$ for the type A of the input terms $t^j_{\mathsf{x}_i}$. $\Diamond$

Lemma 3.12. *The SE rule* predicateAbstrMerge *is strongly exhaustive.*

Proof. Let $s' = \left(C', \mathcal{U}', p\right)$ be an output of an application of predicateAbstrMerge. We have to show that, for a structure K with interpretation function I, concrete states $\sigma, \sigma' \in \mathcal{S}$ and $j = 1, 2$, $\sigma' \in \mathit{concr}_K(s_j, \sigma)$ implies that there is a conservative extension K' of K with interpretation function I' and $\sigma'' \in \mathcal{S}$ s.t. $\sigma' \in \mathit{concr}_{K'}(s', \sigma'')$. We show that this is the case for $\sigma'' := \sigma$ and K' defined as K apart from the interpretation of the new Skolem constants c_{x_i}, for which we set $I'(c_{\mathsf{x}_i}) := \mathit{val}\left(K, \sigma | t^j_{\mathsf{x}_i}\right)$ (same choice as for disjunctionMerge). Since $\sigma' \in \mathit{concr}_K(s_j, \sigma)$, it holds that $K, \sigma \models C_j$, and, where $\sigma''' = \mathit{val}\left(K, \sigma | \mathcal{U}_j\right)(\sigma)$, $(\sigma''', \sigma') \in \varrho(p)$. Because the Skolem constants are introduced freshly, they do not occur in C_j and we also have $K', \sigma \models \mathbb{C}_j$. The constraints on the constants are also satisfied: Since c_{x_i} evaluates in K' the same as $t^j_{\mathsf{x}_i}$, it holds by Item (2) of Def. 3.16 that $K', \sigma \models \alpha(C_j, t^j_{\mathsf{x}_i})(c_{\mathsf{x}_i})$. Due to the definition of $\preceq$ and by Item (3), it also holds that $K', \sigma \models (\alpha(C_1, t^1_{\mathsf{x}_i}) \sqcup \alpha(C_2, t^2_{\mathsf{x}_i}))(c_{\mathsf{x}_i})$. Furthermore, we also have $\mathit{val}\left(K', \sigma | \mathcal{U}'\right)(\sigma) = \sigma'''$, since the Skolem constants evaluate in K' exactly to the values of the right-hand sides in $\mathcal{U}_j$ or, if a x_i is not present in $\mathcal{U}_j$, to the value of the variable itself. It follows that $\sigma' \in \mathit{concr}_{K'}(s', \sigma)$. $\Box$

Concluding this section, we show a simple state merging rule which, in contrast to those presented before, is not exhaustive, but in exchange precise: The rule, branchSelMerge, selects one of the input branches and discards the other one. It is the only rule for which it plays no role whether the input states have the same program counters; also, the structure of the symbolic stores is irrelevant. Following the definition, we state the precision lemma for branchSelMerge, but omit the—obvious—formal proof.

Definition 3.18 (The Branch Selection Merge Rule)**.** Let, for $i = 1, 2$, s_i be two SESs.

Then, the SE rule branchSelMerge is defined as

$$\text{branchSelMerge}\ \frac{s_i}{s_1 \quad s_2}(i \in \{1,2\})$$

◊

Lemma 3.13. *The SE rule* branchSelMerge *is strongly precise.*

3.3.1 Specifications for State Merging

At the time of publishing Reference [SHB16], state merging in KeY could only be applied *interactively* during proof search: When a long proof does not finish in time, it is suitably pruned by the user such that eligible goals have the same program counter. Then, the generic state merging rule is applied on one of the goals. In the appearing dialog, a concrete merge rule (e.g., predicate abstraction) and merge partner goals are selected. Optionally, one can also enter a precise separating formula. If the chosen merge rule requires additional parameters, a follow-up dialog specifically implemented for the particular merge rule appears. In the case of predicate abstraction, that dialog asks for a specification of "core" predicates and a combinator which defines the lattice structure. There are three combinators: Construct lattice elements of all conjunctions, of all disjunctions, or take a flat lattice of the predicates only (plus top/bottom elements).

This approach is usually fine in larger proofs which are, anyway, frequently interaction-heavy. However, it has disadvantages: When restarting SE, e.g., because of an adapted postcondition or changed program, the generic merge rule has to be instantiated from scratch. This is especially cumbersome for predicate abstraction, since the predicates have to be specified again. Furthermore, interactive merging is quite intransparent. It is not visible in the code (as opposed to auxiliary specifications for, e.g., loops) and difficult to read from the proof tree. Also, old proofs are not always loadable in newer KeY versions due to broken backwards compatibility after major changes to the prover.

To overcome both problems, we implemented *Merge Point Statements (MPSs)* in KeY.[5] This feature has only been documented in a blog post[6] before. The idea is to add *in source*

[5] An idea with the same goal was pursued in a Bachelor's thesis [Men17] supervised by the author of this thesis, where *merge block contracts* are proposed and implemented. MPSs are an independent implementation with a different "look-and-feel", which is available in KeY's master branch.

[6] `https://www.key-project.org/2017/05/03/new-feature-state-merging-in-key/`

code an annotation (the MPS) at the point where states shall be merged if possible. The MPS contains specifications describing how to instantiate the generic merge rule.

We demonstrate this along an example. Listing 3.1 shows a method computing the Greatest Common Divisor (GCD) of two integers `a` and `b`. It uses a helper `gcdHelp` which we do not show. The contract of `gcdHelp` ensures that it returns the GCD of two numbers if they are nonnegative and the first is greater or equal than the second. This example is interesting for state merging since before calling `gcdHelp`, to which most work is delegated, `gcd` performs three normalizations ensuring the precondition of `gcdHelp`: It inverts parameter `a` if it is negative, and similarly for `b` afterward. Then, it swaps the two if `a` is smaller or equal than `b`. As these normalizations, especially the two leading inversions, occur early and cause small changes, we can save a comparatively large amount of proof steps by quickly bringing the branches back together.

We added two MPSs to the method, one after each conditional inversion of the inputs (Lines 10 and 15). After each such **`merge_point`** declaration, a **`merge_proc`** specification of a concrete merge rule type follows. In case of the first MPS, this is If-Then-Else Merging (Line 11), in case of the second Predicate Abstraction (Line 16). For these specifications, one needs to know the pre-defined name of the merge rule to use. There are two options apart from the ones in Listing 3.1: “MergeByIfThenElseAntecedent” (which corresponds to pathCondMerge), and “MergeByFullAnonymization”, where the symbolic state is erased by fresh constants. The default is “MergeByIfThenElse”, whose specification can therefore be omitted. The predicate abstraction rule is parametrized. In Line 17, we specify the **`merge_params`** for the rule: We chose a conjunctive combination of two predicates (written in lambda notation). The first predicate represents all positive integers, the second both the original and negated original value of method parameter `b`. Since all states reaching that MPS will only differ in `b`, this is a very precise abstraction which, in addition, also asserts that `b` is positive at that position in the source code.

SE of this method in KeY will, fully automatically, apply state merging twice. Moreover, it is precisely documented in the source code which merge rules were used in the most recent proof. For predicate abstraction, KeY automatically tries to pick the most specific sound abstraction from the given predicates. To that end, it starts a background proof. If that proof fails and a too coarse abstraction is picked, one has the option to resort to an interactive application of the merge rule, where a particular choice can be enforced. Then, it has to be proven in a side branch of the main proof that the abstraction is sound.

Listing 3.1: Merge Point Statements: GCD Example

```
/*@ public normal_behavior
  @   ensures (a != 0 || b != 0) ==>
  @           (a % \result == 0 && b % \result == 0 &&
  @            (\forall int x; x > 0 && a % x == 0 && b % x == 0;
  @                              \result % x == 0));
  @*/
public static int gcd(int a, int b) {
    if (a < 0) a = -a;

    //@ merge_point
    //@ merge_proc "MergeByIfThenElse"; // optional

    if (b < 0) b = -b;

    //@ merge_point
    //@ merge_proc "MergeByPredicateAbstraction"
    //@ merge_params {
    //@   conjunctive:
    //@   (int x) -> {x >= 0, (x == \old(b) || x == -\old(b))}
    //@ };

    int big, small;
    if (a > b) {
        big = a;
        small = b;
    }
    else {
        big = b;
        small = a;
    }

    return gcdHelp(big, small);
}
```

3.3.2 The TimSort Case Study

As one of the largest case studies carried out so far with KeY, de Gouw et al. verified the sorting routine for non-trivial types implemented in the OpenJDK (called "TimSort" after its inventor, Tim Peters). In [Gou+15], they reported on a bug discovered during the first verification attempts. In that publication, two (symmetric) methods, mergeLo and mergeHi, could not be proven; they were out of reach due to state explosion. The proofs in that case study were not conducted naively; for instance, *block contracts* were used to mitigate state explosion. However, they did not always have the desired effect.

The follow-up journal version [Gou+19], among other things, describes how state merging could be used to prove the correctness of these methods in KeY after all. The proof of mergeHi was performed using a clever combination of 36 interactive cuts and 5 applications of state merging. In total, it demanded 460,409 rule applications, of which 3,312 were interactive (not counting rule applications which afterward were pruned). For the journal paper, also the proofs for other methods were repeated, and state merging could be used for seven methods. Initially (as for mergeHi and mergeLo), If-Then-Else merging was used, which lead to an average proof size reduction of 39%, and 80% reduction in case of one larger method (mergeAt).

For two methods, the proof size grew with If-Then-Else-based state merging; for one of those, a proof size *reduction* of 15% could still be accomplished when using predicate abstraction instead. From the experience of using state merging in a complex case study, the authors draw the following conclusions: Using If-Then-Else-based state merging is most beneficial when (1) the program splits early, and a relatively large number of statements follows after the split, and (2) the merged states do not differ too much. Otherwise, the more complex expressions introduced by state merging can lead to an overhead which is not compensated by the reduced amount of SE steps. We refer to [Gou+19] for a more detailed report and concrete figures.

3.4 Summary and Discussion

We defined a universal framework for Symbolic Execution. Our definition is based on the central notion of *concretization* of SESs to concrete states: Given a satisfying interpretation of abstract symbols in the path condition of an SES, we interpret its symbolic store based on a concrete initial state, and transform the result according to the program counter. Since we include the program counter, it is possible to symbolically execute a program without changing the semantics of the SESs *at one level*. For instance, $\big(\emptyset, \mathit{Skip}, \texttt{\textbf{if} (x<0) x=-x;}\big)$

represents all concrete states where $\mathtt{x}$ is nonnegative. Its successors, $(\mathtt{x} < 0, \mathtt{x} := -\mathtt{x})$ and $(\mathtt{x} \geq 0, \mathit{Skip})$, *together* represent the same set. The use of "together" is central to our framework: We describe SE transitions as transformations between SE *configurations*, which are *sets* of SESs. In the example, the initial state forms a singleton configuration, while the final configuration consists of the two successor states.

Transitions are defined m-to-n, comprising traditional 1-to-n as well as m-to-1 transitions common for state merging. There are two traditions of SE: *Lightweight* SE used for test generation and bug finding, and *heavyweight* SE used for program proving. We define two general correctness notions for SE, *precision* and *exhaustiveness*, and relate them to lightweight and heavyweight SE. Thus, we do not exclude any of the two traditions. Because additionally, we support m-to-n steps of *arbitrary granularity*, our framework is extremely flexible and general. We dedicate one section solely to state merging, which the theoretic framework natively supports. Previous approaches formally treating SE [Kne91; LRA17; BB19] do not consider state merging specifically.

One shortcoming of our approach is that we do not enforce a *progress property*. For instance, a transition from the two-state configuration mentioned above *back* to the singleton configuration containing the original state is both precise and exhaustive. This is not allowed in simulation-based frameworks. We decided, in favor of generality, against a definition enforcing progress. However, we think that as future work, a modular *progress property* should be defined. It would also be interesting to define progress properties for different degrees of granularity, e.g., to distinguish single and multi-step SE relations.

4 Abstract Execution

Program proving "is the process of turning the correctness of a program into a mathematical statement and then proving it" [Fil11]. Thanks to the growing power of automated first-order provers (such as Vampire [RV99], Alt-Ergo [Bob+08], CVC3 [BT07] and Z3 [MB08]) and the continuing evolution of dedicated program provers (like Frama-C [Cuo+12], KLEE [CDE08], Java PathFinder [PV04], KeY [Ahr+16], and others), it is now possible to prove complex properties about programs in industrial programming languages such as C, C# and Java [Alk+10; Kle+10; PL10; Gou+19]—at high cost: For the micro kernel verified in [Kle+10], creating the "abstract specification" took 4 person months, while the cost for the proof was about 11 person *years*. Admittedly, this project was about medium-sized, real C code, and relied on an *interactive* theorem prover. Reference [Gou+15] reports on the verification of the sorting algorithm of the Java standard library for non-trivial types, "TimSort". This project used KeY, i.e., a *semi-automatic* tool on a complex, but smaller piece of code. Still, the specification of all and verification of all but two methods of the algorithm took three person-months. Only *re*doing those proofs (for a journal version [Gou+19]) based on the already existing specifications took around one person week, although less than 1 % of all rule applications required user interaction. Also [Kle+10] contains an experience report on the *costs of change*: The authors took 1.5–2 person *years* to re-verify their code after adding new, cross-cutting features to the kernel.

While notably, a fully verified and usable micro kernel is a remarkable result, and the verification attempts for TimSort unveiled a reproducible, subtle bug [Gou+15], functional verification is still too expensive to apply on an "every-day" piece of code. Even after successful verification, the result only holds for the particular verified program; changing it will invalidate the result. One obvious way to mitigate this problem is to invest in *automation*, and to avoid the use of interactive provers whenever possible [Fil11]. Addressing the *specification bottleneck* [Bau+12], i.e., the lack of contracts and auxiliary specifications that are expensive to come up with, it is appealing to look into *generic* properties (not requiring manual specifications) or, e.g., *relational verification* (where a program serves as a specification for another one) instead of full functional verification [Ahr+16].

We generalize the latter suggestion: The necessary cost for deductive program verification is better invested *when the verification result applies not only to one, but to a whole class of programs*. Such second-order properties are usually even more difficult to verify. In exchange, they have an increased relevance, and might indeed be applicable to the mentioned "every-day" piece of code. Consider the following example:

Kiki, a professional programmer, reviews the implementation of a formally verified, safety-critical software project and encounters the following code snippet:

```
... if (y>0) {x=1; ...} else {x=1; ...} ...
```

To increase code quality, she changes that code to

```
... x=1; if (y>0) {...} else {...} ...
```

by pulling the assignment to the variable `x` before the conditional. It is easy to see that this change does not affect the semantics of the program, since the guard of the **`if`** statement does not depend on `x` and none of the involved expressions has a critical side effect, e.g., throws an exception. Let now p be the original and p' be the changed program. To verify that p' is correct, she can prove that a *functional specification*, say, some property $\mathit{Post}(\mathtt{x},\mathtt{y})$ about the variables `x` and `y`, is still satisfied. This can be expressed as $\mathit{Pre} \rightarrow [p']\mathit{Post}(\mathtt{x},\mathtt{y})$ in JavaDL. Alternatively, she can show the *relational* property $\mathit{Pre} \wedge \mathtt{x} \doteq \mathtt{x'} \wedge \mathtt{y} \doteq \mathtt{y'} \rightarrow [p(\mathtt{x},\mathtt{y})][p'(\mathtt{x'},\mathtt{y'})](\mathtt{x} \doteq \mathtt{x'} \wedge \mathtt{y} \doteq \mathtt{y'})$, (where p, p' contain exactly the variables denoted in parentheses, and `x`/`x'` and `y`/`y'` are different program variables). In either case, it constitutes a problem when the "..." contain large pieces of code; creating verification conditions then can be very time consuming. The situation gets worse if Kiki finds out that one of her colleagues committed the same flaw several times in the project, such that she has to invest a lot of time for verifying the code base after performing structurally similar changes throughout different places. Instead, she could save much time by *once and for all* proving that the following equivalence holds (under certain conditions) for all programs q and guards g in arbitrary contexts:

$$\begin{array}{l} \quad\;\, \mathbf{if}\ (g)\ \{q\ \ldots\}\ \mathbf{else}\ \{q\ \ldots\} \\ \equiv q\ \mathbf{if}\ (g)\ \{\ldots\}\ \mathbf{else}\ \{\ldots\} \end{array}$$

Then, instead of laboriously verifying several functional or relational properties for different concrete programs, it is sufficient to show that the concrete piece of code that has been, or should be, changed, satisfies the mentioned "certain conditions". For example, program q should not change locations read by guard g. Showing this is significantly easier than performing functional or relational verification of whole programs.

In the example, the program q and guard g are *abstract* program elements representing many different concrete programs and guards, such as, e.g., the assignment `x=1;` and guard `y>0` above. In our formalisms, *concrete* program proving, be it of functional, relational or generic properties, aims to establish that $\models \varphi$ holds, for a formula $\varphi \in \mathrm{Fml}$ containing potentially multiple modalities with concrete programs. *Abstract program proving*, on the other hand, is concerned with *second-order* properties (quantifying over programs) of the form $\forall^{\mathrm{II}} q : A; \psi$ or $\exists^{\mathrm{II}} q : A; \psi$, where A is one of the types *Stmt* of statements and *Expr* of expressions, and $\forall^{\mathrm{II}}$ / $\exists^{\mathrm{II}}$ are JavaDL$^{\mathrm{II}}$ quantifiers (cf. Sect. 2.5 for an introduction to JavaDL$^{\mathrm{II}}$). Proving *existential* second-order properties is, from a proof-theoretic perspective, simple: To prove $\exists q; \psi(q)$ valid, come up with a concrete instantiation q^0 of q and show that $\psi(q^0)$ is valid. Admittedly, it can be arbitrarily difficult to construct such a q^0, even if we know it exists. *Universal* properties $\forall q; \psi(q)$ are usually proven by *structural induction* [Bur74] over the abstract syntax of the target programming language. Early work on proving universal second-order properties about programs relied on pen-and-paper proofs [Lon72; MP67]. Recently, interactive theorem provers are used to mechanize correctness proofs, e.g., in the verified compiler projects CompCert [Ler09] and CakeML [Kum+14], or in the already mentioned example of the seL4 microkernel [Kle+10]. The main drawback of the latter approach is, as pointed out before, the very high effort required to mechanize a programming language and to perform interactive proofs.

In this thesis, we propose a static software analysis principle called *Abstract Execution (AE)* which allows to *automatically* prove a certain class of second-order universal properties about programs. We consider sequential Java programs in the subset supported by the KeY system and JavaDL; the principles of AE are, however, equally applicable to other sequential languages. In a nutshell, the aim of AE is to provide symbolic execution rules for program placeholders (formally, JavaDL$^{\mathrm{II}}$ second-order program constants) and thus circumvent the necessity of performing structural induction. The core idea is to not consider the *structure* of potential instantiations for abstract Java programs at all, but only their *behavior*. Consider the following example from Sect. 2.5:

$$\vdash \langle q_0 \rangle \mathrm{true} \rightarrow \langle q_0 \;\; \texttt{x=1;} \rangle (\mathtt{x} \doteq 1)$$

To prove this, it is not necessary to distinguish the cases where q_0 is an assignment, **`if`** statement or loop. Instead, all one needs to know is that q_0 completes normally and any change to `x` is overwritten by the concrete assignment at the end of the program. In the following variation of the example,

$$\vdash \langle q_0 \rangle \, \mathrm{true} \rightarrow \langle \texttt{x=1;}\ \ q_0 \rangle (\texttt{x} \doteq 1)$$

the only additional information we need is that q_0 does not assign `x`.[1]

By restricting the analysis to consider only the abstract input/output behavior of abstract program constants (i.e., their *frame* and *footprint*) as well as the conditions under which they complete abruptly (e.g., throw an exception), we realize limited second-order inference over programs in terms of *first-order deduction*, yielding an *automatic* approach. Technically, the main building blocks of our solution are:

(1) *abstract state changes* (second-order updates) to represent the effects of second-order program constants like q_0 on local variables and the heap,

(2) creation of separate SE branches for all reasons of abrupt completion of an APE,

(3) *over-approximation* of returned values and thrown exceptions by symbols created "dependently fresh" for identifiers of abstract program placeholders, i.e., they are created fresh on the first occurrence of a placeholder in a program, but are *re-used* every time that this placeholder re-occurs,

(4) a *specification language* to describe the abstract input/output behavior of abstract program constants, as well as preconditions for abrupt, and postconditions for normal and abrupt completion.

Applications of Abstract Execution AE is applicable to many problems involving reasoning about abstract programs. It can be instantiated to (at least) the following tasks: (1) Execution of abstract method calls [BHP14], a special case of AE; (2) automatic soundness proofs of program transformations [SH19a] or of (3) rule-based compilation [SH18] or of (4) of derived SE rules [BRR08]; (5) sound, automatic ("lazy") Symbolic Execution over programs with loops and calls; (6) incremental program development and synthesis [SGF10] (7) abstract resource analysis; and many more. In Chapter 6, we apply AE on program transformation—task (2). We study refactoring rules as described in Martin Fowler's well-known books [Fow99; Fow18]. We model refactoring techniques as abstract programs, formalize preconditions for safe applications of a refactoring technique, and prove behavioral equivalence of the original and refactored version under these assump-

[1] Of course, the formula would also hold if we knew that q_0 assigned exactly the value 1 to `x`. Although the specification of the (partial) concrete outcome of a second-order program constant was not an initial design goal of AE, we also support this through a postcondition mechanism.

tions. In the application of Modal Trace Logic to formalize program verification problems (Chapter 5), we use AE for problems concerned with code transformations. Other applications of AE, including compilation, lazy SE, Correctness-by-Construction using AE, abstract resource analysis, and the application of AE to prove the correctness of refactorings used in parallelization optimization, are discussed in Chapter 8.

Structure of This Chapter Abstract Execution is about abstract programs. In Sect. 4.1, we describe the concrete syntax for writing abstract programs containing placeholder symbols for statements and expressions. Furthermore, we introduce our specification language for refining the semantics of these placeholders. Sect. 4.2 introduces syntax and semantics of our logic for AE, which is an extension of (first-order) JavaDL by abstract updates and abstract placeholders in programs. SE rules for abstract placeholders and simplification rules for abstract updates are presented in Sect. 4.3. Finally, we explain relevant implementation details in Sect. 4.4.

4.1 Specifying Abstract Programs

An abstract Java program contains at least one *Abstract Program Element (APE)*. An APE is either an *Abstract Statement (AS)* or *Abstract Expression (AExp)* symbol, which are extensions to the Java language representing the two types of second-order program constants defined in Sect. 2.5. The syntax to declare them is:

\abstract_statement *Ident*;	for abstract statements
\abstract_expression *Type Ident*	for abstract expressions

Abstract Statements / Abstract Expressions can be used inside a Java program wherever a concrete statement / expression can be used. The symbols *Ident* are *identifiers* for an APE. Semantically, every APE with the same identifier occurring in a program or proof represents the same program element (modulo renaming of used locations). Consider, for instance, the following program:

```
if (\abstract_expression boolean expr) {
  \abstract_statement Stmt;
}
```

Table 4.1: Potential Side Effects of Abstract Program Elements

	Abstract Statements	Abstract Expressions
Change values of variables / fields in the context	✓	✓
Declare new local variables	✓	
Throw exception	✓	✓
Return (with no value)	✓	
Return (with a given value)	✓	
Break to label	✓	
Continue to, break from surrounding loop	✓	
Continue to label of surrounding loop	✓	

This abstract program represents *all* concrete **`if`** statements. The APEs may be substituted by concrete program elements accessing and assigning arbitrary fields and local variables which furthermore may have all side effects that statements and expressions can possibly have. Table 4.1 lists the possible side effects per APE type.

This alone is insufficient for expressing interesting second-order properties. For instance, the JavaDL formula from the introduction to this chapter (with an AS replacing the second-order program constant q_0),

$$\langle \texttt{x=1; \textbf{\textbackslash abstract\_statement} S;} \rangle \texttt{x} \doteq 1,$$

is generally not valid, if we do not further constrain Abstract Statement S s.t. instantiations may not complete abruptly or assign variable `x`. To that end, we introduce a specification language for APEs. Its syntax extends JML *block contracts* [Ahr+16; Lan18]. *An APE is the declaration of an APE symbol together with all specification clauses that constrain it.* Specifications essentially have two responsibilities: First, to define the assigned and accessed locations (the frame and footprint) of the APE, and secondly, to define the circumstances under which it will complete abruptly. We provide examples only for the more general case of ASs (AExps cannot break, continue, or return).

Framing The *Frame Problem* is central to *modular* static program verification, where properties about parts of the program shall be preserved in presence of changes to *other* parts. Intuitively, the frame problem is about showing, in addition to a functional property, that "nothing else changes" [BMR95]. As demonstrated above, framing is also crucial in Abstract Execution: If we specify to which locations an APE has access and which it may possibly modify, we refine the set of potential instantiations such that we can prove more interesting properties. The set of locations to which a piece of code may write is usually called its *frame*, and the set of locations on which it depends its *footprint*. Note that both are *upper bounds*. Frame and footprint can be specified in JML using **assignable** and **accessible** clauses, respectively. For the example above, we could have specified

```
//@ accessible x;
//@ assignable \nothing;
\abstract_statement S;
```

such that the AS S now only represents concrete statements with footprint x that may not assign any location. If we assume for a moment that S completes normally, this frame specification is sufficient to prove the postcondition $x \doteq 1$. However, it is not interesting to prove something for a program which completes normally without assigning anything. Instead, we would like to show that our property holds for *all programs but* those assigning the variable x. In other words, we would like to *abstract* from concrete locations to a certain degree, restricting the set of possible instantiations of AS S to those that are not harmful, while at the same time not restricting too much.

Our solution to this problem is to use the *dynamic frame* theory [Kas06; Kas11] which aims at solving the frame problem in the presence of data abstraction. The core idea of dynamic frames is to use "specification variables" instead of concrete locations for specifying frames and footprints. One can then (abstractly) specify, e.g., disjointness of two sets of locations, or that a particular concrete location is (not) part of some set of locations. Dynamic frames have been implemented in the KeY system [SUW11; Ahr+16], where the type **\locset** is used to represent sets of memory locations (see Sect. 2.2.1 for (operations on) the corresponding logic type *LocSet*). The **\locset** type in the KeY system represents sets of *heap* locations (i.e., object fields and array slots). Additionally, we allow *program variables* in **accessible** and **assignable** specifications (in standard KeY, program variables are either ignored or explicitly forbidden in frame specifications). Finally, JML **assignable** clauses specify which locations can be assigned a value *at most*; there is no construct actually *enforcing* the assignment. Notwithstanding, in AE we occasionally *need* to enforce the assignment of a specific location. We extend JML with the **\hasTo**(·) keyword which may be used in **assignable** clauses. To specify that an APE *must* assign a value to x, for instance, we write `//@ \assignable \hasTo(x)`. Table 4.2 (Page 116) lists JML specification constructs for APE framing.

Notation (Simplified Framing Syntax for Examples). We sometimes use an idealized syntax for APEs instead of the implemented syntax expounded in this chapter: The notation $\mathtt{P}(\mathit{assignables} :\approx \mathit{accessibles})$ / $\mathtt{e}(\mathit{assignables} :\approx \mathit{accessibles})$ represents an AS / an AExp with identifier symbol `P` / `e`, frame *assignables* and footprint *accessibles* which, unless otherwise stated, is expected to complete normally. We use capital letters `P`, `Q`, ... for AS identifiers and lower case letters `e`, `f`, ... for AExp identifiers. We distinguish **`\hasTo`** locations by a superscript exclamation mark as in $\mathtt{P}(\mathtt{x}^!, \mathtt{y} :\approx \mathit{accessibles})$. ◊

Remark 4.1 (Considerations on **`\hasTo`** Locations). The concept of the **`\hasTo`** specifier for parts of the frame of an APE sounds simple at first, but is rather difficult to understand at second thought. Consider, for instance, the AS $\mathtt{P}(\mathtt{x}^! :\approx \mathtt{x}, \mathtt{y})$ which has to assign the variable `x`, being allowed to access at most the variables `x` and `y`. The specification of `x` as a **`\hasTo`** location excludes the empty statement from legal instantiations of AS `P`; however, since `x` is in its footprint, the self-assignment `x=x;` is still part of its legal instantiations. As we discuss in Sect. 4.2, standard trace semantics—and JavaDL—cannot, without further modifications, even distinguish the empty statement from a self-assignment, since both leave the state unchanged. Let us, however, consider a scenario where `P` should operate on a temporary variable `tmp` instead of `x`, and is therefore surrounded by a *set* and *reset* statement as follows:

$$\mathtt{tmp=x;\ P(tmp^! :\approx tmp, y);\ x=tmp;}$$

In this context, the self-assignment `tmp=tmp;` as an instantiation of `P` is semantically equivalent to the assignment `tmp=x;` due to the prior initialization of `tmp`. Indeed, we can drop the set statement after applying it to the footprint of `P`:

$$\mathtt{P(tmp^! :\approx x, y);\ x=tmp;}$$

Since we know that `P` has to assign `tmp` (which is not used afterward), this simplifies to

$$\mathtt{P(x^! :\approx x, y);}$$

This simplification could not be conducted if `tmp` was not marked as **`\hasTo`**. In this case, we could not have dropped the *set* statement since there is no known *write-after-write dependency* between the AS and the *set* statement. Similarly, the final simplification step would not be valid as `P` could represent the empty statement and the *reset* statement could not be merged into `P` the way we did, as there is no *read-after-write dependency* between the *reset* statement and `P` which we could be sure about. Note that if `tmp` was not in the footprint of `P`, we could directly drop the, then *ineffective*, *set* statement entirely, without applying it to the footprint before. Summarizing these observations, we conclude that **`\hasTo`** specifications help performing simplifications (which are necessary to prove, in particular, certain relational properties) by externalizing write-after-write and read-after-write dependencies. *Not having* **`\hasTo`***s is sound, but not complete.* ◊

Listing 4.1: Initial Scaffold for Framing Example

```
if (\abstract_expression boolean g) {
  \abstract_statement Q;
  \abstract_statement P1;
} else {
  \abstract_statement Q;
  \abstract_statement P2;
}
```

We continue the example from the chapter's introduction.

Example 4.1 (Framing). Consider the conditional statement

`if` (g) $\{q\ \ldots\}$ **`else`** $\{q\ \ldots\}$

from which we want to "pull out" program q. We formalize this situation as an abstract program, using an Abstract Expression for guard g and Abstract Statements for statement q and the dots. In this example, we assume that all APEs complete normally. The initial scaffold for the abstract program model is depicted in Listing 4.1. For the transformation to be semantics-preserving, we need to enforce that the footprint of guard `g` and the frame of AS `Q` are disjoint, such that pulling out `Q` has no influence on the evaluation of `g`. Furthermore, we have to keep in mind that also the evaluation of `g` might have side effects that could influence the evaluation of `Q`. Therefore, we globally declare **`\locset`** location sets `frameG`, `frameQ`, `footprintG` and `footprintQ` (keyword **`ae_specvars`**) and add as constraints (keyword **`ae_constraint`**) that the respective frames and footprints have to be disjoint (using the **`\disjoint`** keyword). Note that also the *frames* of `g` and `Q` have to be disjoint, since otherwise, one APE could overwrite previously live assignments of the other. The statements `P1` and `P2` do not have to be constrained and may be given the most general footprint and frame, **`\everything`**. As a postcondition, we choose a term $P($**`\value`**$(\mathit{relevantLocs}))$ for some **`\locset`** specification variable *relevantLocs*. The function **`\value`** converts a location set into the values it represents. The resulting, refined model is shown in Listing 4.2. The clauses for `P1`, `P2` could have been omitted, since **`\everything`** is the default case. Concrete instances of this abstract model consist of an **`if`** statement in which the statements in the "then" and "else" block can be partitioned s.t. they share some common statements as prefix; the footprints and frames of this prefix as well as the guard of the **`if`** statement have to satisfy the imposed disjointness constraints.

Table 4.2: JML Constructs for APE Framing

JML Construct	Explanation
accessible *locs*;	Declares the locations in *locs* to be accessible by the APE.
assignable *locs*;	Declares the locations in *locs* to be assignable by the APE.
\hasTo(*loc*)	Modifier specifying that the location *loc* has to be assigned. Only allowed in **assignable** clauses.
\value (*locs*)	Represents the values stored at the locations in *locs*.

Table 4.3: JML Constructs for APE Abrupt Completion Specifications

JML Construct		Explanation
exceptional_behavior	**requires** φ;	Spec. APE throws exc. iff φ holds.
return_behavior	**requires** φ;	Specified AS completes due to a **return** of no value iff φ holds.
return_val_behavior	**requires** φ;	Specified AS completes due to a **return** of a value iff φ holds.
break_behavior	**requires** φ;	Specified AS breaks/continues
continue_behavior	**requires** φ;	during loop execution iff φ holds.
break_behavior (*lbl*)	**requires** φ;	Specified AS breaks/continues to
continue_behavior (*lbl*)	**requires** φ;	the (loop) label *lbl* iff φ holds.

Table 4.4: JML Constructs for Global AE Specifications

JML Construct	Explanation
ae_specvars *type ids*;	Declares AE specification variables *ids* of type *type*.
ae_constraint φ;	Imposes the AE constraint φ.
\formula, **\locset**	Possible types for *type* in **ae_specvars** declarations.
\disjoint	Standard KeY operator for disjointness of **\locset** sets. See [Ahr+16, Chapter 9] for more such operators.
\mutex	Shortcut declaring the arguments mutually exclusive.

Listing 4.2: Abstract Program Model for Framing Example 4.1

```
/*@ ae_specvars \locset footprintG, frameG, footprintQ, frameQ,
  @                      relevantLocs;
  @ ae_constraint \disjoint(footprintG, frameQ);
  @ ae_constraint \disjoint(footprintQ, frameG);
  @ ae_constraint \disjoint(frameQ, frameG);
  @*/

if (
  //@ accessible footprintG;
  //@ assignable frameG;
  \abstract_expression boolean g
) {
  //@ accessible footprintQ;
  //@ assignable frameQ;
  \abstract_statement Q;

  //@ accessible \everything;
  //@ assignable \everything;
  \abstract_statement P1;
} else {
  //@ accessible footprintQ;
  //@ assignable frameQ;
  \abstract_statement Q;

  //@ accessible \everything;
  //@ assignable \everything;
  \abstract_statement P2;
}

// Post Condition: P(\value(relevantLocs))
```

For example, there is no instance where `g` is instantiated to “`x++>0`” and `Q` to “`x--;`”, which even violates *all* constraints. ◊

The keywords **`ae_specvars`** and **`ae_constraint`** are additions to JML for AE (Table 4.4 lists global specification constructs for AE). The keyword **`ae_constraint`** is used for declaring constraints on the instantiations of APEs. When conducting proofs about abstract programs, these constraints can be treated as *axioms*; when checking whether some program is an instance of an abstract program, on the other hand, it has to be *proven* that this program satisfies the constraints. The keyword **`ae_specvars`** facilitates declarations of global specification variables, in this case **`\locset`** abstract location sets, for AE. These variables have to be suitably instantiated when instantiating an abstract program. The difference to *model fields* (see [Ahr+16, Chapter 9]) is that specification variables declared via **`ae_specvars`** are not bound to a particular compilation unit; furthermore, **`\locset`** specification variables can also represent local variables, and not only fields. Declaring symbols using **`ae_specvars`** clauses is equivalent to globally declaring them in a KeY problem file. Note that constraints on specification variables can be declared locally, i.e., they have to be satisfied by instantiations at the point of their occurrence and may refer to local program variables visible in their context.

The keyword **`\value`** is also an AE-specific addition to JML. It “converts” location set terms to the values they represent. Location sets are *rigid*: For instance, the location set consisting of variable `x` refers to exactly this variable of that particular name, and specifically not to the *value* of variable `x` at the point of its occurrence. On the other hand, the expression **`\value`**`(x)`, where `x` is to be interpreted as location set, represents exactly that value. For location sets that are not just program variables, the conversion is more interesting. We discuss **`\value`** in more detail in Sects. 4.2 and 4.3.

Remark 4.2 (Differences to original publication). The specification framework originally sketched in [SH19a] differs in some points from the updated version in this thesis. (1) Framing in [SH19a] is not based on the general theory of *dynamic frames*. The examples there use concrete local variables more frequently; there is only rudimentary support for abstract location sets in the form of specifications of variable sets “declared” by APEs using the keyword **`declares`** which we dropped in this thesis. Also the **`\value`** keyword was added in the generalization to dynamic frames. (2) When an AS in [SH19a] did not have a **`declares`** annotation, instances were not allowed to declare local variables visible to the outside. Here, all ASs may declare local variables, which are part of the normal (dynamic) frame. (3) We use **`ae_constraint`** instead of the JML keyword **`axiom`** employed in [SH19a]. We decided to introduce a special keyword for our purposes to emphasize that while AE constraints are *axioms* when proving properties about abstract programs, they become *proof obligations* when checking concrete instantiations. (4) We introduced the new keyword **`ae_specvars`** for declaring abstract "higher-order" specification variables such as abstract location sets, predicates, or functions. (5) Finally, the "abstract expression" APE type was not present in [SH19a], where only ASs were used. Using an "abstract expression idiom", abstract expressions were replaced by abstract statements and fresh program variables. This is either *unsound*–if the fresh program variables are declared as **`\hasTo`**–or *incomplete* otherwise (cf. Remark 4.10). Additionally, using the idiom makes instance checking of abstract programs more difficult, since one has to "reverse" it before performing the check. Also, it is not always easily possible to employ the idiom, for instance in the case of a loop guard, which has to be evaluated in every iteration. ◊

Remark 4.3 (Redundancy and Order in APE Specifications). Listing 4.2 exhibits a so far unmentioned peculiarity: AS symbol `Q`, which occurs twice in the listing in Lines 15 and 23, also has to be specified at both occurrences. Considering that an APE is the declaration of a symbol *together with all specification clauses constraining it*, that makes perfect sense; however, since all occurrences of an APE in a program should represent the same concrete instantiations, it would also be a bad idea to specify them *differently*. Then, it would be impossible to instantiate them consistently, which "invalidated" the whole abstract program. Still, we cannot generally assign a contract globally to an APE symbol, applying that contract for all occurrences, for two reasons. First, we sometimes wish to treat APEs *parametric* as in Remark 4.1, such that instances of the same program operate on different frames. Second, locations referred to by specifications may have the same name, but still be different entities. Take, for example, the program scaffold

```
if (...) { int x; ... Q(x:≈x,y) } else { int x; ... Q(x:≈x,y) }
```

In that program, both occurrences of `Q` refer to *different* variables `x`. This also frequently emerges when APEs occur in different method contexts. When attaching specifications directly to the APEs, variable names are always bound correctly depending on the context. Additionally, this is the only solution which is technically realizable in a reliable manner; otherwise, we had to replace variables based on their names when assigning "global" specifications to APE symbol occurrences, which is a quite brittle solution. Thus, when saying that all APE occurrences represent the same program, we mean program "patterns" parametric in assignable and accessible locations.

Talking about consistent specification of APEs, one has to be aware that the *order* of locations in **`assignable`** and **`accessible`** clauses also plays a role. An APE can be interpreted as a collection of functions assigning unknown values to the locations in its frame based on the values of the locations in its footprint. The effect of the program `x=1; Q(y:≈x,z)` will generally be different from the effect of `x=2; Q(y:≈x,z)`. Indeed, if `x` is not used afterward in the context, we can represent both programs as `Q(y:≈1,z)` and `Q(y:≈2,z)`. Now, if there were two additional assignments of the variable `z` in front of the occurrences of the AS assigning `z` the value 2 in the one and 1 in the other case, we would end up with `Q(y:≈1,2)` resp. `Q(y:≈2,1)` after simplification. Clearly, we can expect instantiations to have different effects, since they have different parameters. Therefore, it is not sound to reorder, drop, or otherwise alter "parameters" of APEs (i.e., elements of their footprint specification). Similarly, the order of the elements in the *frame* specification matters. We can expect that the program `Q(x,y:≈`*footprint*`)` has the same effect on `x` than `Q(w,z:≈`*footprint*`)` has on `w`; however, the effect of `Q(y,x:≈`*footprint*`)` (with frame elements swapped) on `x` will be different. ◊

Abrupt Completion Specifications Example 4.1 assumes that APEs complete normally. However, APEs `g` and `Q` might complete abruptly for various reasons (cf. Table 4.1); e.g., `g` can throw an exception and `Q` can return, in which case the program completed because of an *exception* before and because of a *return* after pulling out the prefix. In this regard, we only have to extend the already existing specification language by one construct, since JML block contracts already support specification of almost all behaviors, including rarer cases like **`break`**s to a particular label [Lan18]. The one exception is that so far, completion due to a **`return`** statement *without* and *with a given* value are not distinguished. For concrete programs, the context always uniquely determines which one is possible, depending on whether the surrounding method is **`void`** or not. When analyzing abstract pieces of code in isolation, this information has to be made explicit. We therefore extend JML by a **`return_val_behavior`** specification clause. Concretely, we specify, e.g., **`return_val_behavior requires`** φ`;` to bind the return behavior (*of a value*) of an APE to the formula φ. Concrete instantiations of that APE must return a value if, and only if, the formula φ evaluates to true in the current state. Table 4.3 lists the JML constructs used for specifying abrupt completion behavior. The following example continues Example 4.1, adding specifications for abrupt completion to render pulling out the prefix of the **`if`** statement a sound operation.

Example 4.2 (Specification of Abrupt Completion). Consider again Example 4.1. We refine Listing 4.2 s.t. abrupt completion of APEs `g` and `Q` is not ignored, but explicitly controlled to enable pulling out the prefix `Q` of the **`if`** statement. The extended abstract program model is shown in Listing 4.3; we highlighted added lines in gray. We introduce new AE specification variables of type **`\formula`** that are coupled to the exceptional behavior of `g` and `Q` and the return behavior of `Q` by **`requires`** clauses of the corresponding behavior specifications. Furthermore, we add another **ae_constraint** (Lines 9 to 11) which specifies, using the new keyword **\mutex**, that those are *mutually exclusive*. This means that AS Q may not return or throw an exception if AExp g throws an exception. Finally, we have to add two further constraints on the frames of g and Q: In the setting where g completes because of an exception, Q has no chance to change the relevant state (i.e., affect the valuation of the abstract location set *relevantLocs*) in the original version, but it can do so after having been pulled out. The same holds if Q completes abruptly. The constraints in Lines 5 to 6 state that the changes of g and Q to the state are not (directly) relevant. They may still, of course, affect the effects of P1 and P2 on the relevant state. Note that we did not specify the completion of AS Q due to (labeled or unlabeled) **break** and **continue** statements. Instead, we assume that the abstract program fragment occurs outside any loop and in a context without leading labels, and that impossible behavior (like breaking without a loop) is not triggered (cf. Remark 4.9). Otherwise, we would have to add more specification cases and consider them in the **\mutex** expression. ◊

Functional Specifications Although it is not one of the main use cases of Abstract Execution, APEs can also be annotated with functional postconditions (**ensures** clauses). This touches the border between abstract execution and block contracts (see the subsequent Remark 4.4). The effect of postconditions on reasoning with APEs is the same as for loop invariants: In the partially "anonymized" post state after execution of the APE, we can assume the postcondition to hold. Instantiations of APE with postconditions have to ensure those. While for relational verification (the main use case of AE), functional postconditions are usually not needed, they are interesting for applications like *Correctness-by-Construction (CbC)*. We refer to Chapter 8 for a discussion of AE for CbC.

Remark 4.4 (Difference to Block Contracts). The JML fragment used for specifying APEs is very similar to JML block contracts [Ahr+16; Lan18]. In fact, we extend the existing syntax, and also re-use significant parts of the *implementation* of block contracts in KeY (see Sect. 4.4). There is one big difference: Block contracts have *concrete* pre- and postconditions as well as frame conditions, and annotate a *concrete* Java block. In KeY, when symbolically executing a Java block annotated with a block contract, one has the choice of *either* using the contract *or* directly executing the code in the block, ignoring the contract. "*Fully Abstract Operation Contracts*" [BHP14] decouple reasoning about programs from the applicability check of contracts by permitting placeholders in *specifications*. Still, annotated programs are concrete, and the presented method indeed makes most sense on the method level, not at the statement level where AE resides. AE goes a step further: Abstract Program Elements are *fully abstract* by themselves. Additional specification only

Listing 4.3: Abstract Program Model for Abrupt Completion Example 4.2

```
//@ ae_specvars \locset footprintG, frameG, footprintQ, frameQ, relevantLocs;
//@ ae_constraint \disjoint(footprintG, frameQ);
//@ ae_constraint \disjoint(footprintQ, frameG);
//@ ae_constraint \disjoint(frameQ, frameG);
//@ ae_constraint \disjoint(frameG, relevantLocs);
//@ ae_constraint \disjoint(frameQ, relevantLocs);

//@ ae_specvars \formula throwsExcQ(any), returnsQ(any), throwsExcG(any);
/*@ ae_constraint \mutex(throwsExcQ(\value(footprintQ)),
  @                      returnsQ(\value(footprintQ)),
  @                      throwsExcG(\value(footprintG))); */

if (
  //@ accessible footprintG;
  //@ assignable frameG;
  //@ exceptional_behavior requires throwsExcG(\value(footprintG));
  \abstract_expression boolean g
) {
  //@ accessible footprintQ;
  //@ assignable frameQ;
  //@ exceptional_behavior requires throwsExcQ(\value(footprintQ));
  //@ return_behavior requires returnsQ(\value(footprintQ));
  \abstract_statement Q;

  //@ accessible \everything;
  //@ assignable \everything;
  \abstract_statement P1;
} else {
  //@ accessible footprintQ;
  //@ assignable frameQ;
  //@ exceptional_behavior requires throwsExcQ(\value(footprintQ));
  //@ return_behavior requires returnsQ(\value(footprintQ));
  \abstract_statement Q;

  //@ accessible \everything;
  //@ assignable \everything;
  \abstract_statement P2;
}

// Post Condition: P(\value(relevantLocs))
```

can make them more concrete, refining the set of potential instantiations. APE *symbols* can, per definition, not be decoupled from their specifications. In contrast to a code block, we therefore cannot execute an APE "ignoring its specification". $\Diamond$

Subsequently, we expound our logic for AE, which syntactically and semantically represents the specification constructs presented in this section.

4.2 Abstract Execution Logic: Syntax and Semantics

Our logic for AE is realized by extending JavaDL with two main ingredients:

- *Abstract Program Elements (APEs)* that can be used inside Java programs, and
- *Abstract Updates*, which reflect abstract state changes caused by APEs.

In this section, we define the syntax and semantics of these constituents. Sect. 4.3 presents corresponding calculus rules. The central notion for the semantics of AE is that of *legal instantiations*: A Symbolic Execution State (SES) (or sequent) containing any of the above listed abstract elements of the logic represents all concrete SESs (or sequents) resulting from consistent instantiations of the abstract elements respecting all constraints (in particular, framing and behavioral conditions of APEs). This reflects the second-order character of AE, which allows reasoning about properties not only of individual, but of a range of programs. Starting from a formal definition of Abstract Program Elements, we define legal *abstract program fragments* containing APEs and global AE specifications. Next, we characterize frame conditions and behavioral specifications by JavaDL formulas, providing us with the means to formalize legal instantiations of abstract program fragments. Afterward, we introduce *abstract updates*. First, however, we extend the location set theory of JavaDL to support dynamic frames in Abstract Execution. In particular, we use program variables in frame and footprint specifications, which was not intended in the definition of the existing *LocSet* theory (cf. Sect. 2.2).

4.2.1 Extensions of the LocSet Theory

We introduce a new type *ProgVar* with associated infinite domain $\mathscr{D}^{ProgVar}$ of *program variable locations*. An element of this set, similar to a heap location (o, f), "statically" refers to a program variable location of a given name, and not to the value of that variable in a state. The mandatory vocabulary of JavaDL is extended by six function symbols:

$$\begin{aligned}
\mathit{singletonPV} &: \mathrm{PVSym} \to \mathit{ProgVar} \\
\cdot^{!} &: \mathit{LocSet} \to \mathit{LocSet} \\
\mathit{value} &: \mathit{LocSet} \to \mathit{Any} \\
\mathit{heapLocs} &: \mathit{LocSet} \to \mathit{LocSet} \\
\mathit{pvLocs} &: \mathit{LocSet} \to \mathit{LocSet} \\
\mathit{anonPV} &: \mathit{ProgVar} \times \mathit{LocSet} \times \mathit{ProgVar} \to \mathit{ProgVar}
\end{aligned}$$

The *LocSet* constructor *singletonPV* allows to include local variables in location set definitions. A *LocSet* expression $\mathit{set}^{!}$ represents the same locations as *set*;[2] the function has the purpose of marking locations that have to be overwritten (see Remark 4.1). The semantics of a term *value*(*set*) are the values attained by the locations represented by the location set *set*. For instance, the meaning of $\mathit{value}(\mathit{singletonPV}(\mathtt{x}))$ is the value of program variable x in the current state. The functions *heapLocs* and *pvLocs* are filters for heap and program variable location sets, respectively. A term $\mathit{anonPV}(\mathit{singletonPV}(\mathtt{x}), \mathit{locset}, \mathit{singletonPV}(\mathtt{x}'))$ evaluates to the program variable location $\mathit{singletonPV}(\mathtt{x}')$ if $\mathit{singletonPV}(\mathtt{x})$ is in *locset* and to $\mathit{singletonPV}(\mathtt{x})$ otherwise. This construct is used for conditional anonymization of program variables, similar to *anon* for heaps.

We extend the definition of $\mathcal{D}^{\mathit{LocSet}}$ to include program variable locations:

$$\mathcal{D}^{\mathit{LocSet}} := \text{the set of all subsets of } \{(o, f) \mid o \in \mathcal{D}^{\mathit{Object}}, f \in \mathcal{D}^{\mathit{Field}}\} \cup \mathcal{D}^{\mathit{ProgVar}}$$

Consequently, we adapt the semantics of *allLocs* to represent this new domain:

$$\begin{aligned}
\mathit{val}(K, \sigma|\mathit{allLocs}) :=& (\mathit{val}(K, \sigma|\mathit{Object}) \times \mathit{val}(K, \sigma|\mathit{Field})) \cup \\
& \mathit{val}(K, \sigma|\mathit{ProgVar})
\end{aligned}$$

The bijective function $\mathit{val}(K, \sigma|\mathit{singletonPV})$ maps program variable symbols to their corresponding elements in $\mathit{val}(K, \sigma|\mathit{ProgVar})$. It is *interpreted*, i.e., $\mathit{val}(K, \sigma|\mathit{singletonPV})$ evaluates to the same function in all structures K. For a program variable *symbol* $\mathtt{x} \in$ PVSym, we write $\underline{\mathtt{x}}$ for the program variable *location* $\mathit{val}(K, \sigma|\mathit{singletonPV})(\mathtt{x}) \in \mathcal{D}^{\mathit{ProgVar}}$. The semantics of the remaining extensions is defined as follows:

$$\mathit{val}(K, \sigma|\circ^{!})(\mathit{set}) := \mathit{set}$$

[2] Indeed, a rule transforming a term $t^{!}$ to t is sound. The purpose of $\cdot^{!}$ becomes clearer in the context of abstract updates (Sect. 4.2.3). Removing the $\cdot^{!}$ in the left-hand side of an abstract update is not possible, since it is part of the operator itself, and does not form a term.

$$
\begin{aligned}
val(K,\sigma|value)(set) &:= \{\sigma(\mathsf{heap})(o,f) \mid (o,f) \in set \cap (\mathcal{D}^{Object} \times \mathcal{D}^{Field})\} \cup \\
&\quad\ \{\sigma(\mathsf{x}) \mid \underline{\mathsf{x}} \in set \cap \mathcal{D}^{ProgVar}\} \\
val(K,\sigma|heapLocs)(set) &:= set \cap (\mathcal{D}^{Object} \times \mathcal{D}^{Field}) \\
val(K,\sigma|pvLocs)(set) &:= set \cap \mathcal{D}^{ProgVar} \\
val(K,\sigma|anonPV)(\underline{\mathsf{x}}, set, \underline{\mathsf{x}}') &:= \begin{cases} \underline{\mathsf{x}}' & \text{if } \underline{\mathsf{x}} \in set \\ \underline{\mathsf{x}} & \text{otherwise} \end{cases}
\end{aligned}
$$

Furthermore, the predicate ε is adapted such that it can be used with program variable locations: $(\underline{\mathsf{x}}, set) \in val(K,\sigma|\varepsilon)$ iff $\underline{\mathsf{x}} \in set$.

Notation (Simplified Notation for Extended Location Set Theory). We use some simplified notation for (lists of) location sets. Instead of $singletonPV(\mathsf{x})$, we frequently write $\mathring{\mathsf{x}}$ for brevity. We write $loc \in set$ equivalently for $\varepsilon(loc, set)$ (where loc is a program variable location or an object-field pair). For a list $\overrightarrow{sets} = (set_1, set_2, \ldots, set_n)$, we write $value(\overrightarrow{sets})$ for the tuple $(value(set_1), \ldots, value(set_n))$. Apart from that, at all places where we use a list of *LocSet* terms where a single one should appear, we generally mean the union of these terms. For example, the meaning of $loc \in \overrightarrow{sets}$ is $\varepsilon(loc, union(set_1, union(set_2, union(\ldots, set_n))))$. Instead of the latter expression, we also write $loc \in \bigcup\{set_1, set_2, \ldots, set_n\}$. ◊

4.2.2 Syntax and Semantics of APEs and Abstract Program Fragments

We regard APEs as tuples of (1) an *identifier*, (2) a *type*, where ASs have the designated pseudo type STATEMENT, (3) a *frame* and (4) *footprint* specification, (5) a *termination specifier*, which is one of the values PARTIAL (APE has to terminate) and TOTAL (APE may diverge), and (6) a set of *specifications* especially for sufficient and necessary preconditions of abrupt completion behavior. Specifications can also have postconditions, i.e., a specification for, e.g., abrupt completion due to a **`return`** is a pair consisting of a precondition and a postcondition. In relational verification, the postcondition is frequently omitted and thus logically equals true. Normal completion does not have a precondition in AE; an APE completes normally if, and only if, it does not complete abruptly. As before, we write "APE P" short for "the APE with identifier symbol P". For easing the notation, we continue to identify APEs with their identifier symbols in formulas; also, we use the parameter notation "P(*assignables* :≈ *accessibles*)" as before to make the frames and footprints of APEs explicit. Subsequently, we formally define the logic representation of APEs.

Definition 4.1 (Abstract Program Elements)**.** An Abstract Program Element is a tuple

$$(id, type, assignables, accessibles, term, specs)$$

of an *identifier id*, a *type type* $\in$ TSym $\cup$ {STATEMENT}, a *frame specification assignables* $\in (\mathrm{Trm}_{LocSet})^n$, $n \geq 1$, a *footprint specification accessibles* $\in (\mathrm{Trm}_{LocSet})^m$, $m \geq 1$, a *termination specifier term* $\in$ {PARTIAL, TOTAL} and *behavioral specifications specs*. The latter is a tuple of the form

$$\begin{aligned}(&normalPost,\\ &returnsSpec, returnsValSpec, excSpec, continuesSpec, breaksSpec,\\ &continuesSpecLbl, breaksSpecLbl)\end{aligned}$$

where

- $normalPost \in \mathrm{Fml}$,
- the elements
  $$\begin{aligned}&returnsSpec, returnsValSpec, excSpec, continuesSpec, breaksSpec \in\\ &\{(Pre, Post) \mid Pre, Post \in \mathrm{Fml}\}\end{aligned}$$
  are pairs of formulas defining pre- and postconditions for abrupt completion of the an APE due to a **`return`** of a value and of no value, exception, **`continue`**, and **`break`**, respectively,
- The elements
  $$continuesSpecLbl, breaksSpecLbl : Labels \rightharpoonup \mathrm{Fml} \times \mathrm{Fml}$$
  are partial functions from Java labels to pairs of pre- and postconditions for abrupt completion due to a labeled **`continue`** or labeled **`break`**,
- all preconditions are mutually exclusive, i.e., for any two preconditions Pre_1 and Pre_2, it holds that $Pre_1 \land Pre_2$ is not satisfiable,
- pre- and postconditions may contain local program variables of the context as well as the following special program variables:
  - `heap` : *Heap*, to access heap locations,
  - $\mathtt{heap}^{pre}$ to access heap locations in the state before the APE was executed,
  - `exc` : Exception, to refer to the exception in the case that the APE completes abruptly due to a thrown exception (postcondition of *excSpec* only),
  - `res` : Object, to refer to the result value returned by an AS (postcondition of *returnsValSpec* only), ◊

Abstract program fragments are program fragments containing at least one APE, along with global declarations of AE specification variables (**`ae_specvars`**) and constraints on them (**`ae_constraint`**). They define the domains of the specification elements *continuesSpecLbl* and *breaksSpecLbl*: APEs have to supply pre- and postconditions for exactly the labels in the context of their appearance in the abstract program fragment.

Remark 4.5 (Compulsory Specification of Pre- and Postconditions). The specification of pre- and postconditions for all reasons of abrupt completion of an APE is mandatory. This is practical for defining the semantics of APEs and abstract program fragments. In practice, it would however be inconvenient to always write complete specifications, and furthermore, one frequently wants to permit abrupt completion without specific constraints. To mitigate the latter issue, one can annotate an APE with "abstract" pre- and trivial postconditions, for instance in the case of exceptional completion with terms $pre_{ape}^{exc}(value(accessibles))$ as precondition and "true" as postcondition, for predicate symbols pre_{ape}^{exc} that are created fresh, but are re-used for each APE occurrence with the same identifier symbol. These function symbols would then be added as AE specification variables, along with corresponding constraints assuring mutual exclusion with the other reasons for abrupt completion (cf. the abstract **`\formula`** specification variables and corresponding constraints in Example 4.2). To address the former issue, we can create those abstract pre- and postconditions *automatically*, which is exactly what we do in practice. ◊

Constraints can also be declared *locally* within an abstract program fragment s.t. they can refer to globally unavailable locations, e.g., the exception variable of a **`catch`** clause. Nevertheless, they are w.l.o.g. treated globally in the following definition of abstract program fragments: Local constraints can be converted to global ones by interpreting them in the SES of their occurrence. We distinguish two types of specification variables: Abstract location sets (for dynamic frames and footprints), and abstract function and predicate symbols used in the abstract specification of the behavior of APEs.

Definition 4.2 (Abstract Program Fragments). Let *Prg* be a (concrete) Java program. An *abstract program fragment* in the context of *Prg* is a tuple

$$(p, APEs, locSpecVars, funcAndPredSymbols, constraints)$$

where

- p is a sequence of statements containing exactly the APEs in the non-empty set *APEs*,
- *locSpecVars* is a set of dynamic frame specification variables, i.e., constant symbols

of type *LocSet*, comprising at least the symbols used in the APEs in *APEs*,

- *funcAndPredSymbols* $\subseteq$ FSym $\cup$ PSym is a set of function and constant symbols comprising all otherwise undefined symbols used in pre- and postconditions of the *specs* element of the APEs,
- *constraints* $\subseteq$ Fml is a set of constraints on *locSpecVars* and other abstract specification elements defining the behavior of the APEs,
- all $ape_1, ape_2 \in APEs$ *with the same identifier* have lists of frame specifications of the same lengths (similarly for footprints),
- for all $ape \in APEs$, it holds that $dom(continuesSpecLbl)$ and $dom(breaksSpecLbl)$ coincide with the (loop) labels in the context of the occurrence of *ape* in p. ◊

A program fragment is a *legal instantiation* of an *abstract* program fragment if APEs and AE specification variables are substituted by concrete statements, expressions, and locations such that the resulting program fragment is legal and the substitutions satisfy all declared constraints, i.e., frame conditions, behavioral APE constraints and constraints on specification variables. We formalize this as JavaDL formulas. This is practical for checking whether a concrete program instantiates an abstract one. To show that a concrete statement respects, for instance, the frame condition of an AS, one can then show the corresponding JavaDL formula. Subsequently, we only consider statements. All proof obligations can also be applied for expressions e of type T by considering the statement T `x=`e`;` for a fresh variable `x` that is added to the frame specification for that statement (obviously, AExps can only be instantiated by concrete expressions of a subtype of the type T of the AExp). For all proof obligations, we assume that the implicit premise "$wellFormed(\mathtt{heap}) \rightarrow \ldots$" is prepended if it is not explicitly included.

Frame and Footprint Let *assignables* be a specification of assignable locations and p be a Java statement. To satisfy the *frame condition*, p must comply with two conditions: (1) it must *at most* assign locations in *assignables* (the "classic" frame condition), (2) it must *at least* assign the locations in $assignables_{hasTo} := \{loc : loc^! \in assignables\}$ ("has-to condition"). To satisfy the *footprint condition*, p must *at most* read from locations in *accessibles*.

(1) Frame Condition To satisfy the frame specification, p must at most modify the locations defined by the *LocSet* expressions *assignables*. This is ensured by the validity of the following formula, where $\mathtt{x}_1, \dots, \mathtt{x}_n$ are all program variables occurring in p:

$$
\begin{aligned}
&\mathit{frameFor}(\mathit{assignables}, p) := \\
&\quad \{\mathsf{heap}^{pre} := \mathsf{heap} \,\|\, \mathtt{x}_1^{pre} := \mathtt{x}_1 \,\|\, \dots \,\|\, \mathtt{x}_n^{pre} := \mathtt{x}_n\} \\
&\qquad [p]\big((\forall f : \mathit{Field}; \forall o : \mathit{Object}; \\
&\qquad\qquad o.\mathit{created}@\mathsf{heap}^{pre} \doteq \mathrm{FALSE} \vee o.f \doteq o.f@\mathsf{heap}^{pre} \\
&\qquad\qquad\quad \vee (o, f) \in \mathit{assignables}) \wedge \\
&\qquad\quad (\mathtt{x}_1 \doteq \mathtt{x}_1^{pre} \vee \mathring{\mathtt{x}}_1 \in \mathit{assignables}) \wedge \dots \wedge (\mathtt{x}_n \doteq \mathtt{x}_n^{pre} \vee \mathring{\mathtt{x}}_n \in \mathit{assignables})\big)
\end{aligned}
$$

(2) has-to Condition Standard JavaDL is not capable of recording memory access; e.g., the program `x=x;` and the *empty* statement "`;`" are equivalent w.r.t. the Java transition relation ϱ and their trace semantics (the state is left unchanged). Therefore, we cannot express that at least the locations in $\mathit{assignables}_{\mathit{hasTo}}$ are assigned—unless we extend the logic suitably. Recently, an extension of JavaDL has been proposed to track accessed memory locations in a global history [BHH19]. With this extension, we could express the semantics of the has-to condition as a JavaDL formula $\mathit{hasToFor}(\mathit{assignables}, p)$. Since it would go beyond the scope of this thesis, we abstain from introducing this theory here. When checking whether a statement instantiates an AS, we *conservatively overapproximate* $\mathit{hasToFor}(\mathit{assignables}, p)$ by a static check: It evaluates to tt if each location in $\mathit{assignables}_{\mathit{hasTo}}$ occurs *literally* as a left-hand side in an assignment in p (this includes shorthand notations like `x++` which unfold to `x=x+1`). Otherwise, we let it evaluate to ff.

(3) Footprint Specification Program p satisfies the footprint (*accessibles*) specification if the evaluation in two *arbitrary* environments which agree on the values of the accessible locations yields the same effects on the assignable locations. Following the *dependency contracts* framework in [SUW11], we can rephrase the condition as "if we change all locations except for *accessibles* in an unknown way, then this must not affect the evaluation of p on the locations in *assignables*", which we formalize as follows:

$$
\begin{aligned}
&\mathit{footprintFor}(\mathit{accessibles}, \mathit{assignables}, p) := \\
&\quad \mathit{wellFormed}(\mathsf{heap}) \wedge \mathit{wellFormed}(h) \rightarrow \\
&\qquad \big([p]\mathit{Post}(\mathit{value}(\mathit{assignables})) \leftrightarrow \\
&\qquad\quad \{\mathsf{heap} := \mathit{anon}(\mathsf{heap}, \mathit{setMinus}(\mathit{allLocs}, \mathit{heapLocs}(\mathit{accessibles})), h) \,\| \\
&\qquad\quad\; \mathtt{x}_1 := \mathit{value}(\mathit{anonPV}(\mathring{\mathtt{x}}_1, \mathit{setMinus}(\mathit{allLocs}, \mathit{pvLocs}(\mathit{accessibles})), \mathring{\mathtt{x}}_1^a)) \,\| \dots \|
\end{aligned}
$$

$$\mathtt{x}_n := value(anonPV(\mathring{\mathtt{x}}_n, setMinus(allLocs, pvLocs(accessibles)), \mathring{\mathtt{x}}_n^a))\}$$
$$[p]Post(value(assignables))\big)$$

where

- *Post* is a fresh predicate of suitable type and arity,
- the $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are all program variables occurring in p,
- h is a fresh *Heap* symbol and $\mathtt{x}_i^a$ are fresh program variables of the types of $\mathtt{x}_i$.

In contrast to the dependency contracts setting (see [SUW11] and [Ahr+16, Sect. 8.3.2]), we are not only interested in the effects on one particular variable (the result variable), but on a whole *set* of locations. Therefore, our post condition contains all locations of interest, which are those that are assignable by an APE.

(4) Termination An APE specifies whether instances must terminate or may also diverge depending on the value of the *term* field. Concretely, instances must terminate if $term =$ TOTAL, which is formalized straightforwardly with the diamond modality (note, however, that this is undecidable due to the Halting Problem):

$$terminationFor(term, p) := term = \text{TOTAL} \rightarrow$$
$$\langle \texttt{try } \{p\} \texttt{ catch (Throwable t) } \{\}\rangle \text{true}$$

We have to wrap p in a **`try`** block since otherwise, the formula would not evaluate to tt if p threw an exception, in spite of the liberal postcondition “true”, due to the semantics of the diamond modality: If p throws an exception, however, it terminates, and therefore *does* meet the termination requirement.

Behavioral Constraints To capture compliance with behavioral constraints, we use *completion scopes* (see Sect. 2.4). The idea is to wrap the program in a completion scope and to record relevant completion information in the branches of the scope.

(5) Normal Completion A legal instance of an APE has to complete normally when none of the preconditions for abrupt completion is met. In this case, it also has to satisfy the postcondition for normal termination. The precondition for normal termination is captured by the following formula *normal*, where the function *pre* extracts the precondition of a pre- and postcondition pair:

$$
\begin{aligned}
\mathit{normal} := {} & \neg \mathit{pre}(\mathit{returnsSpec}) \land \neg \mathit{pre}(\mathit{returnsValSpec}) \land \neg \mathit{pre}(\mathit{excSpec}) \\
& \land \neg \mathit{pre}(\mathit{continuesSpec}) \land \neg \mathit{pre}(\mathit{breaksSpec}) \\
& \land \bigwedge_{l \in \mathit{dom}(\mathit{continuesSpecLbl})} \neg \mathit{pre}(\mathit{continuesSpecLbl}(l)) \\
& \land \bigwedge_{l \in \mathit{dom}(\mathit{breaksSpecLbl})} \neg \mathit{pre}(\mathit{breaksSpecLbl}(l))
\end{aligned}
$$

In the following formalization of the normal completion requirement, the boolean variable `_normal` is fresh, and label l does not occur in p. The variable `_normal` is initialized to **`false`** and set to **`true`** if p terminates normally and thus the added labeled **`break`** statement is reached. If p terminates abruptly for any other reason, `_normal` stays **`false`** and the postcondition cannot be proven. Otherwise, also the postcondition *normalPost* for the normal completion case has to hold for the statement to be true.

$$
\begin{aligned}
& \mathit{normalCompletionFor}(\mathit{specs}, p) := \mathit{normal} \rightarrow \\
& \quad [\texttt{\_normal=}\textbf{\texttt{false}}\texttt{;} \\
& \quad\; \textbf{\texttt{exec}}\ \{p\ \textbf{\texttt{break}}\ l\texttt{;}\}\ \textbf{\texttt{ccatch}}\ (\textbf{\texttt{\textbackslash Break}}\ l)\ \{\texttt{\_normal=}\textbf{\texttt{true}}\texttt{;}\}] \\
& \quad\; (\texttt{\_normal} \doteq \mathrm{TRUE} \land \mathit{normalPost})
\end{aligned}
$$

The formalizations for the remaining completion types work analogously, but are simpler: For normal completion, we had to add the artificial labeled **`break`** statement only because there is no **`ccatch`** clause for normal completion. Additionally to the function *pre* for extracting preconditions of pairs, we use the function *post* for postconditions below.

(6) Completion due to Return We distinguish completion due to a **`return`** with and without a value. The case for **`return`** without a value is simpler:

$$
\begin{aligned}
& \mathit{returnsFor}(\mathit{returnsSpec}, p) := \mathit{pre}(\mathit{returnsSpec}) \rightarrow \\
& \quad [\texttt{\_returned=}\textbf{\texttt{false}}\texttt{;}
\end{aligned}
$$

$$\texttt{exec}\ \{p\}\ \texttt{ccatch}\ (\texttt{\textbackslash Return})\ \{\texttt{\_returned=true;}\}]$$
$$(\texttt{\_returned} \doteq \mathrm{TRUE} \land post(returnsSpec))$$

For a **return** with a given value, we also have to assign the special program variable `res` (see Def. 4.1), such that the returned value is available in *returnsValSpec*.

$$
\begin{array}{l}
returnsValFor(returnsValSpec, p) := pre(returnsValSpec) \rightarrow \\
\quad [\texttt{\_returned=false;} \\
\quad\ \ \texttt{exec}\ \{p\} \\
\quad\ \ \texttt{ccatch}\ (\texttt{\textbackslash Return val})\ \{\texttt{res=val; \_returned=true;}\}] \\
\quad\ \ \ (\texttt{\_returned} \doteq \mathrm{TRUE} \land post(returnsValSpec))
\end{array}
$$

(7) Exceptional Completion The exceptional case is similar to **return** of a value; we only have to make the `exc` variable available to the postcondition instead of `res`.

$$
\begin{array}{l}
excFor(excSpec, p) := pre(excSpec) \rightarrow \\
\quad [\texttt{\_didThrow=false;} \\
\quad\ \ \texttt{exec}\ \{p\}\ \texttt{ccatch}\ (\texttt{Throwable t})\ \{\texttt{exc=t; \_didThrow=true;}\}] \\
\quad\ \ \ (\texttt{\_didThrow} \doteq \mathrm{TRUE} \land post(excSpec))
\end{array}
$$

(8) Completion due to Continue and Break Those cases are similar to the case of a **return** without a value:

$$
\begin{array}{l}
breaksFor(breaksSpec, p) := pre(breaksSpec) \rightarrow \\
\quad [\texttt{\_didBreak=false;} \\
\quad\ \ \texttt{exec}\ \{p\}\ \texttt{ccatch}\ (\texttt{\textbackslash Break})\ \{\texttt{\_didBreak=true;}\}] \\
\quad\ \ \ (\texttt{\_didBreak} \doteq \mathrm{TRUE} \land post(breaksSpec))
\end{array}
$$

$$
\begin{aligned}
&\mathit{continuesFor}(\mathit{continuesSpec}, p) := \mathit{pre}(\mathit{continuesSpec}) \rightarrow \\
&\quad [\texttt{\_didContinue=}\textbf{\texttt{false}}\texttt{;} \\
&\quad\ \textbf{\texttt{exec}}\ \{p\}\ \textbf{\texttt{ccatch}}\ (\texttt{\textbackslash}\textbf{\texttt{Continue}})\ \{\texttt{\_didContinue=}\textbf{\texttt{true}}\texttt{;}\}] \\
&\quad\ \ (\texttt{\_didContinue} \doteq \mathrm{TRUE} \land \mathit{post}(\mathit{continuesSpec}))
\end{aligned}
$$

(9) Completion due to Labeled Continue and Break For the last two cases dealing with completion due to *labeled* **`continue`** and **`break`** statements, we need one proof obligation for each label in the domains of *continuesSpecLbl* and *breaksSpecLbl*, respectively, since all those are associated to separate pre- and postconditions.

$$
\begin{aligned}
&\mathit{breaksForLbl}(\mathit{breaksSpecLbl}, \mathit{lb}, p) := \mathit{pre}(\mathit{breaksSpecLbl}(\mathit{lb})) \rightarrow \\
&\quad [\texttt{\_didBreak=}\textbf{\texttt{false}}\texttt{;} \\
&\quad\ \textbf{\texttt{exec}}\ \{p\}\ \textbf{\texttt{ccatch}}\ (\texttt{\textbackslash}\textbf{\texttt{Break}}\ \mathit{lb})\ \{\texttt{\_didBreak=}\textbf{\texttt{true}}\texttt{;}\}] \\
&\quad\ \ (\texttt{\_didBreak} \doteq \mathrm{TRUE} \land \mathit{post}(\mathit{breaksSpecLbl}(\mathit{lb})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{continuesForLbl}(\mathit{continuesSpecLbl}, \mathit{lb}, p) := \mathit{pre}(\mathit{continuesSpecLbl}(\mathit{lb})) \rightarrow \\
&\quad [\texttt{\_didContinue=}\textbf{\texttt{false}}\texttt{;} \\
&\quad\ \textbf{\texttt{exec}}\ \{p\}\ \textbf{\texttt{ccatch}}\ (\texttt{\textbackslash}\textbf{\texttt{Continue}}\ \mathit{lb})\ \{\texttt{\_didContinue=}\textbf{\texttt{true}}\texttt{;}\}] \\
&\quad\ \ \big(\texttt{\_didContinue} \doteq \mathrm{TRUE} \land \mathit{post}\,(\mathit{continuesSpecLbl}(\mathit{lb}))\big)
\end{aligned}
$$

We combine all formulas introduced above into a single formula *represents*(ape, p) which evaluates to *tt* if the program p is a legal instance of (is represented by) the APE *ape*:

$$
\begin{aligned}
&\mathit{represents}(\mathit{ape}, p) := \\
&\quad\ \ \mathit{frameFor}(\mathit{assignables}, p) \\
&\quad \land \mathit{hasToFor}(\mathit{assignables}, p) \\
&\quad \land \mathit{footprintFor}(\mathit{accessibles}, \mathit{assignables}, p) \\
&\quad \land \mathit{terminationFor}(\mathit{term}, p)
\end{aligned}
$$

$$
\begin{aligned}
&\wedge\ \mathit{normalCompletionFor}(\mathit{specs}, p)\\
&\wedge\ \mathit{returnsFor}(\mathit{returnsSpec}, p)\\
&\wedge\ \mathit{returnsValFor}(\mathit{returnsValSpec}, p)\\
&\wedge\ \mathit{excFor}(\mathit{excSpec}, p)\\
&\wedge\ \mathit{breaksFor}(\mathit{breaksSpec}, p)\\
&\wedge\ \mathit{continuesFor}(\mathit{continuesSpec}, p)\\
&\wedge \bigwedge_{\mathit{dom}(\mathit{breaksSpecLbl})} \mathit{breaksForLbl}(\mathit{breaksSpecLbl}, \mathit{lb}, p)\\
&\wedge \bigwedge_{\mathit{dom}(\mathit{continuesSpecLbl})} \mathit{continuesForLbl}(\mathit{continuesSpecLbl}, \mathit{lb}, p)
\end{aligned}
$$

Based on *represents*, we can define the semantics of a *single* APE as follows.

Definition 4.3 (Semantics of APEs)**.** Let *abstrStmt* be an AS. Its semantics $[\![\mathit{abstrStmt}]\!]$ is the set of all concrete statements represented by it, formally:

$$[\![\mathit{abstrStmt}]\!] := \{\mathit{stmt} : \mathit{Stmt} \mid\ \models \mathit{represents}(\mathit{abstrStmt}, \mathit{stmt})\}$$

The definition works accordingly for AExps. ◊

Remark 4.6 (Correctness of Semantics for APEs). Usually, it is strictly speaking not meaningful to discuss the "correctness" of the definition of the semantics of logical structures. The semantics defines the meaning of the logic symbols, such that one can later on speak about soundness and completeness of syntactical operations on those symbols. A semantics can, in this regard, not be "wrong"; it can only fail to faithfully capture our intuition. Our definition of the semantics of APEs comes in a special flavor, though, based on the validity of JavaDL formulas created from an APE. It would therefore be incorrect if, for example, it did *not* hold for a concrete statement p respecting its specifications *specs* that p completes normally if, and only if, $\mathit{normalCompletionFor}(\mathit{specs}, p)$ is valid. We omit the corresponding formal proofs showing that this is the case, in the firm conviction that a proof is only valuable as long as it helps to thoroughly understand the problem statement.[3] For the formulas defining the semantics of APEs, which are, regarded in isolation, sufficiently small and clear, we do not think that this is the case. In the following, we assume that these formulas evaluate accordingly to their intended meaning. ◊

[3] This does not apply to mechanized proofs; such proofs are valuable as they are especially trustworthy. Notwithstanding, for mechanized proofs it is crucial to validate—on an informal "meta" level—that the *formalizations* are faithful.

Legal instantiations of *abstract program fragments* have to first provide instantiations of the AE specification variables satisfying the global constraints; second, they have to provide legal *and consistent* instantiations of the APEs s.t. the resulting program is a legal program fragment in the non-abstract sense. An instantiation of a set of APEs is consistent if two APEs with the same identifier are instantiated by programs which at most differ in the *names* of the locations they refer to, but are otherwise equivalent. This becomes most obvious when using the parameter notation for APEs: Two occurrences $\mathtt{P}(\mathtt{x} :\approx \mathtt{y})$ and $\mathtt{P}(\mathtt{z} :\approx \mathtt{w})$ may by instantiated with the statements "`x=2*y;`" and "`z=2*w;`", but not with "`x=y;`" and "`z=2*w;`". In other words, all instantiations for APEs with the same identifier must expose the same externally observable behavior w.r.t. a bijective mapping between the actually used locations. We subsequently define the notion of a *behavioral program isomorphism* formalizing this intuition; an example is given afterward.

Definition 4.4 (Behavioral Program Isomorphism)**.** Let p_1, p_2 be two legal Java program fragments and frame_i / $\mathit{footprint}_i \in \mathcal{D}^{\mathit{LocSet}}$ their frames and footprints. For structures K and functions $f : \mathcal{D}^{\mathit{LocSet}} \to \mathcal{D}^{\mathit{LocSet}}$, we define the equivalence relation $\equiv_f^K \subseteq \mathcal{S} \times \mathcal{S}$ such that $\sigma \equiv_f^K \sigma'$ iff $\mathit{val}(K, \sigma|\mathit{value}(\mathit{loc})) = \mathit{val}(K, \sigma'|\mathit{value}(f(\mathit{loc})))$ for all $\mathit{loc} \in \mathit{dom}(f)$. A *Behavioral Program Isomorphism* for p_1, p_2 is a bijective mapping ι between $\mathit{frame}_1 \cup \mathit{footprint}_1$ and $\mathit{frame}_2 \cup \mathit{footprint}_2$ such that for all structures K and states $\sigma_2 \equiv_\iota^K \sigma_1$, it holds that $(\sigma_1, \sigma_1') \in \varrho(p_1)$ if, and only if, $(\sigma_2, \sigma_2') \in \varrho(p_2)$ and $\sigma_2' \equiv_\iota^K \sigma_1'$. Two programs are *behaviorally isomorphic*, written $p_1 \approxeq_\iota p_2$, if ι is a behavioral program isomorphism for them. ◊

Example 4.3 (Behavioral Program Isomorphism)**.** Let K be a structure and value^K the valuation of function symbol *value* in K. Consider the programs $p_1 :=$ `x=2*y;` and $p_2 :=$ `z=2*w;` from above. They are behaviorally isomorphic: Let ι be such that $\iota(\underline{\mathtt{x}}) = \underline{\mathtt{z}}$ and $\iota(\underline{\mathtt{y}}) = \underline{\mathtt{w}}$. All transitions in $\varrho(p_1)$ have the form $(\sigma, \sigma[\mathtt{x} \mapsto 2\sigma(\mathtt{y})])$. For $\varrho(p_2)$, they have the form $(\sigma', \sigma'[\mathtt{z} \mapsto 2\sigma'(\mathtt{w})])$ which comprises $(\sigma, \sigma[\iota(\underline{\mathtt{x}}) \mapsto 2\sigma(\mathit{value}^K(\iota(\underline{\mathtt{y}})))])$. Program p_1 and the program $p_3 :=$ `z=2*z;` are *not* behaviorally isomorphic since the variable sets have different cardinalities. Also, p_1 and $p_4 :=$ `z=2*w+2;` are not behaviorally isomorphic, since the transitions for p_4, which are of the form $(\sigma', \sigma'[\mathtt{z} \mapsto 2\sigma'(\mathtt{w}) + 2])$, do not comprise $(\sigma, \sigma[\iota(\underline{\mathtt{x}}) \mapsto 2\sigma(\mathit{value}^K(\iota(\underline{\mathtt{y}})))])$. Note, however, that p_1 and $p_2' :=$ `z=w;z+=z;` *are* behaviorally isomorphic, as the set

$$\varrho(p_2') = \{(\sigma, \sigma'') \mid \sigma' = \sigma[\mathtt{z} \mapsto \sigma(\mathtt{w})], \sigma'' = \sigma'[\mathtt{z} \mapsto \sigma'(z) + \sigma'(\mathtt{z})]\}$$

is equivalent to

$$\{(\sigma, \sigma'') \mid \sigma' = \sigma[\mathtt{z} \mapsto \sigma(\mathtt{w})], \sigma'' = \sigma'[\mathtt{z} \mapsto 2 * \sigma'(\mathtt{z})]\}$$

and therefore ultimately to $\varrho(p_2) = \{(\sigma, \sigma[\mathtt{z} \mapsto 2 * \sigma(\mathtt{w})]\})$, and p_2 is behaviorally isomorphic to p_1. This demonstrates that behavioral isomorphism is weaker than *syntactic* isomorphism, as we cannot obtain p_1 by renaming locations in the syntax of p_2'. As a final example, consider $p_2'' := \{\texttt{int k=w;z=2*k;}\}$, for which

$$\varrho(p_2'') = \{(\sigma, \sigma'' \mid \sigma'' = \sigma'[\mathtt{z} \mapsto \sigma'(\mathtt{2k})], \sigma' = \sigma[\mathtt{k} \mapsto \sigma(\mathtt{w})])\}$$

which is equivalent to $\{(\sigma, \sigma[\mathtt{k} \mapsto \sigma(\mathtt{w})][\mathtt{z} \mapsto \sigma(\mathtt{2w})]\}$. This allows us to derive that also p_2'' is behaviorally isomorphic to p_1, since the variable $\mathtt{k}$ *is contained neither in the (external) frame nor in the footprint of* p_2'', because it is declared inside a block and not visible to the outside, and thus not in $dom(\iota)$. Behavioral program isomorphism is agnostic to locally scoped variable declarations. $\Diamond$

We continue with the definition of legal instantiations of abstract program fragments. In the following, the notation $S[subst]$ denominates the result of applying the substitution *subst* on all elements of the set S; similarly for programs p instead of sets.

Definition 4.5 (Legal Instantiations and Semantics of Abstract Program Fragments)**.** Let $\mathscr{F} = (p, APEs, locSpecVars, funcAndPredSymbols, constraints)$ be an abstract program fragment. A legal Java program fragment p^0 is a *legal instantiation* of $\mathscr{F}$ if it arises from a substitution $subst_{locSpecVars}$ of concrete locations for specification variables, a substitution $subst_{funcAndPredSymbols}$ for abstract function and predicate symbols, as well as an instantiation $subst_{APEs}$ of concrete statements and expressions for APEs s.t.

(1) $dom(subst_{locSpecVars}) = locSpecVars$, $dom(subst_{funcAndPredSymbols}) = funcAndPredSymbols$ and $dom(subst_{APEs}) = APEs$,

(2) $p^0 = p[subst_{APEs}]$,

(3) the formulas $constraints[subst_{funcAndPredSymbols}][subst_{locSpecVars}]$ are valid (i.e., the global constraints on AE specification variables are satisfied),

(4) each instantiation in $subst_{APEs}$ is *represented* by the APE it instantiates, respecting the instantiations of specification variables: For all $ape \in APEs$, it holds that

$$ape[subst_{APEs}] \in [\![ape[subst_{funcAndPredSymbols}][subst_{locSpecVars}]]\!],$$

(5) and the instantiation $subst_{APEs}$ is *consistent*: for all APEs ape_1, $ape_2 \in APEs$ with the same identifier symbol, it holds that $ape_1[subst_{APEs}] \simeq_\iota ape_2[subst_{APEs}]$, for a mapping ι respecting the order of frame and footprint specifications in ape_1 and ape_2. Formally: If, for $i = 1, 2$, $\overrightarrow{fr}_i \in (\mathrm{Trm}_{LocSet})^n$, $\overrightarrow{fp}_i \in (\mathrm{Trm}_{LocSet})^m$ are the tuples of frame and footprint specifiers for ape_i, $\overrightarrow{locs_i} \in (\mathrm{Trm}_{LocSet})^{(n+m)}$ is the concatenation

of these tuples for ape_i, and, for $j = 1, \dots, n+m$, $set_i^j \in \mathcal{D}^{LocSet}$ the instantiation of $\overrightarrow{locs}_i(j)$, there have to be bijective mappings $\iota_1, \dots, \iota_{n+m}$ between $set_1^j \in \mathcal{D}^{LocSet}$ and $set_2^j \in \mathcal{D}^{LocSet}$ that can be combined *without conflicts* to a behavioral isomorphism ι for $ape_1[subst_{APEs}]$ and $ape_2[subst_{APEs}]$.

The semantics $[\![\mathcal{F}]\!]$ of $\mathcal{F}$ is then defined as the set of its legal instantiations. ◊

Two mappings can be "combined without conflicts" if they either have disjoint domains or map equal elements to the same result. For instance, two mappings ι_1 and ι_2 with $\iota_1(\underline{\mathtt{x}}) = \underline{\mathtt{z}}$ and $\iota_2(\underline{\mathtt{x}}) = \underline{\mathtt{w}}$ cannot be combined without conflicts.

Remark 4.7 (Instance Checking for Abstract Program Fragments). Many applications of AE require verifying that a program is an instance of an abstract program fragment. This can only be practical if the conditions defining the property of being a legal instance can be expressed *syntactically*, such that they can in principle be statically proven. Def. 4.5 permits such an approach: Items (1) and (2) are purely syntactic constraints. Item (3) consists in proving JavaDL formulas (without APEs). Item (4) refers to the condition of Def. 4.3, which is already expressed in terms of a syntactic JavaDL formula. The mapping ι demanded by Item (5) can be also be created syntactically by associating the instantiations for frame and footprint elements of APEs. This can, alternatively, also be overapproximated by admitting instantiations that are equal up to a bijective renaming of their used locations. That we can express the property of being a legal instantiation syntactically does of course not mean that we can decide it; for instance, the proof might require to prove termination (an instance of the undecidable halting problem). Still, we can automatically prove this property in many cases: It is, for example, trivial to prove that a program fragment terminates if it does not contain loops nor method calls. ◊

Example 4.4 (Abstract Program Fragments and Legal Instantiations). Consider the abstract program in Listing 4.4. It contains two ASs with identifier symbols `A` and `B` with "independent" frames and footprints and mutually exclusive abrupt completion behavior. We assume that all completion modes that are not covered by an appropriate behavior clause default to "**`requires false;`**". The program models the pre-state of a "Slide Statements" refactoring: Due to the imposed constraints, we can swap the statements and maintain the postcondition relation that returned values, thrown exception objects or otherwise the whole state are equivalent before and after swapping the statements. Formally, this abstract program fragment is represented by a tuple

$$\mathcal{F} = \big(p, \{A, B\}, \{frameA, footprintA, frameB, footprintB\},$$

Listing 4.4: Abstract Program Model for Example 4.4

```
/*@ ae_constraint
  @   \disjoint(frameA, frameB) &&
  @   \disjoint(frameA, footprintB) &&
  @   \disjoint(frameB, footprintA) &&
  @
  @   \mutex(returnsA(\value(footprintA)), returnsB(\value(footprintB))) &&
  @   \mutex(returnsA(\value(footprintA)), throwsExcB(\value(footprintB))) &&
  @   \mutex(throwsExcA(\value(footprintA)), throwsExcB(\value(footprintB))) &&
  @   \mutex(throwsExcA(\value(footprintA)), returnsB(\value(footprintB)));
  @*/

//@ assignable frameA;
//@ accessible footprintA;
//@ exceptional_behavior requires throwsExcA(\value(footprintA));
//@ return_val_behavior requires returnsA(\value(footprintA));
\abstract_statement A;

//@ assignable frameB;
//@ accessible footprintB;
//@ exceptional_behavior requires throwsExcB(\value(footprintB));
//@ return_val_behavior requires returnsB(\value(footprintB));
\abstract_statement B;
```

$$\big\{returnsValA(Any), throwsExcA(Any), returnsValB(Any), throwsExcB(Any)\big\},$$
$$\{disjoint(frameA, frameB), disjoint(frameA, footprintB), \dots\}\big)$$

where p is the program `A;B;` consisting of the ASs A and B. The Abstract Statements are defined as follows:

$$\begin{aligned} A = \big(\mathsf{A}, \textsc{statement}, (frameA), (footprintA), \textsc{total},\\ (\text{true}, (\text{false}, \text{true}),\\ (returnsValA(value(footprintA)), \text{true}),\\ (throwsExcA(value(footprintA)), \text{true}),\\ (\text{false}, \text{true}), (\text{false}, \text{true}), \emptyset, \emptyset)\big) \end{aligned}$$

$$\begin{aligned} B = \big(\mathsf{B}, \textsc{statement}, (frameB), (footprintB), \textsc{total},\\ (\text{true}, (\text{false}, \text{true}),\\ (returnsValB(value(footprintB)), \text{true}),\\ (throwsExcB(value(footprintB)), \text{true}),\\ (\text{false}, \text{true}), (\text{false}, \text{true}), \emptyset, \emptyset)\big) \end{aligned}$$

Note that all postconditions default to “true”, and that, since there are no labels in the context of the ASs, the partial functions for specifications of completion due to labeled **break**s and **continue**s are undefined on all arguments (written as an empty set).

To show that, e.g., the concrete program $p^0 :=$ `x=z; z*=2; i=2k;` is a legal instance of $\mathscr{F}$, i.e., that $p^0 \in [\![\mathscr{F}]\!]$, we need to find suitable instantiations for the specification variables and APEs such that the requirements of Def. 4.5 are met. To that end, let

$$\begin{aligned} subst_{locSpecVars} :=\ & \{frameA \mapsto \{\mathring{\mathtt{x}}, \mathring{\mathtt{z}}\}, footprintA \mapsto \{\mathring{\mathtt{z}}\},\\ & frameB \mapsto \{\mathring{\mathtt{i}}\}, footprintB \mapsto \{\mathring{\mathtt{k}}\}\}\\ subst_{funcAndPredSymbols} :=\ & \{returnsValA \mapsto \text{false}, throwsExcA \mapsto \text{false},\\ & returnsValB \mapsto \text{false}, throwsExcB \mapsto \text{false}\}\\ subst_{APEs} :=\ & \{A \mapsto \mathtt{x=z;z*=2;}, B \mapsto \mathtt{i=2k;}\} \end{aligned}$$

Requirements (1) to (2) are obviously satisfied: The substitutions have the right domains and ranges, and substituting the APEs in p yields p^0. Item (3) corresponds to proving the instantiations of the abstract constraints declared in Lines 1 to 10 in Listing 4.4. After substitution of the concrete frames, the first constraint, for example, becomes $\mathit{disjoint}(\{\mathring{\mathtt{x}}, \mathring{\mathtt{z}}\}, \{\mathring{\mathtt{i}}\})$, and the very last one $\mathit{mutex}(\text{false}, \text{false})$. Both are valid (due to the definition of *mutex*, it must not be the case that both arguments are true, which is not the case here). To show requirement (4), we have to verify that the two instantiations for APEs A and B are legal instances, i.e., that both $\mathit{represents}(A, \mathtt{x=z;z*=2;})$ and $\mathit{represents}(B, \mathtt{i=2k;})$ hold. This corresponds to proving two lengthy conjunctions of formulas. We consider frame and footprint conditions and the condition for normal completion for AS A as examples. For the frame condition, we obtain

$$\begin{aligned}
&\mathit{frameFor}(\mathit{frameA}, \mathtt{x=z;z*=2;})[\mathit{subst}_{\mathit{funcAndPredSymbols}}][\mathit{subst}_{\mathit{locSpecVars}}] = \\
&\quad \{\mathsf{heap}^{pre} := \mathsf{heap} \,||\, \mathsf{x}^{pre} := \mathsf{x} \,||\, \mathsf{z}^{pre} := \mathsf{z}\} \\
&\qquad [\mathtt{x=z;z*=2;}]\big((\forall f : \mathit{Field}; \forall o : \mathit{Object}; \\
&\qquad\qquad o.\mathit{created}@\mathsf{heap}^{pre} \doteq \mathrm{FALSE} \lor o.f \doteq o.f\,@\mathsf{heap}^{pre} \lor (o, f) \in \{\mathring{\mathsf{x}}, \mathring{\mathsf{z}}\}) \land \\
&\qquad\quad (\mathsf{x} \doteq \mathsf{x}^{pre} \lor \mathring{\mathsf{x}} \in \{\mathring{\mathsf{x}}, \mathring{\mathsf{z}}\}) \land (\mathsf{z} \doteq \mathsf{z}^{pre} \lor \mathring{\mathsf{z}} \in \{\mathring{\mathsf{x}}, \mathring{\mathsf{z}}\})\big)
\end{aligned}$$

which is a valid formula since the program does not change the heap (i.e., $o.f \doteq o.f\,@\mathsf{heap}^{pre}$ is true for all objects and fields) and all occurring program variables are frame elements (i.e., $\mathring{\mathsf{x}} \in \{\mathring{\mathsf{x}}, \mathring{\mathsf{z}}\}$ and $\mathring{\mathsf{z}} \in \{\mathring{\mathsf{x}}, \mathring{\mathsf{z}}\}$ holds).

The footprint condition has the following shape (recall that the fresh predicate *Post* contains the specified frame locations):

$$\begin{aligned}
&\mathit{footprintFor}(\mathit{footprintA}, \mathit{frameA}, \mathtt{x=z;z*=2;})[\mathit{subst}_{\mathit{funcAndPredSymbols}}][\mathit{subst}_{\mathit{locSpecVars}}] = \\
&\quad \mathit{wellFormed}(\mathsf{heap}) \land \mathit{wellFormed}(h) \to \\
&\qquad \big([\mathtt{x=z;z*=2;}]\mathit{Post}(\mathit{value}(\mathring{\mathsf{x}}), \mathit{value}(\mathring{\mathsf{z}})) \leftrightarrow \\
&\qquad\quad \{\mathsf{heap} := \mathit{anon}(\mathsf{heap}, \mathit{setMinus}(\mathit{allLocs}, \mathit{heapLocs}(\{\mathring{\mathsf{z}}\})), h) \,|| \\
&\qquad\quad\; \mathsf{x} := \mathit{value}(\mathit{anonPV}(\mathring{\mathsf{x}}, \mathit{setMinus}(\mathit{allLocs}, (\{\mathring{\mathsf{z}}\})), \mathring{\mathsf{x}}^{a})) \,|| \\
&\qquad\quad\; \mathsf{z} := \mathit{value}(\mathit{anonPV}(\mathring{\mathsf{z}}, \mathit{setMinus}(\mathit{allLocs}, (\{\mathring{\mathsf{z}}\})), \mathring{\mathsf{z}}^{a}))\} \\
&\qquad\qquad [\mathtt{x=z;z*=2;}]\mathit{Post}(\mathit{value}(\mathring{\mathsf{x}}), \mathit{value}(\mathring{\mathsf{z}}))\big)
\end{aligned}$$

Also this formula is valid. Anonymizing the heap has no effect here. The anonymization $\mathsf{x} := \mathit{value}(\mathit{anonPV}(\mathring{\mathsf{x}}, \mathit{setMinus}(\mathit{allLocs}, (\{\mathring{\mathsf{z}}\})), \mathring{\mathsf{x}}^{a}))$ evaluates to the fresh ("anonymous") program variable x^{a}, which has no effect since the program indeed respects its footprint condition. The expression $\mathit{value}(\mathit{anonPV}(\mathring{\mathsf{z}}, \mathit{setMinus}(\mathit{allLocs}, (\{\mathring{\mathsf{z}}\})), \mathring{\mathsf{z}}^{a}))$, however, evaluates to z, since $\mathring{\mathsf{z}}$ is *not* in the set of all locations *but* itself. Therefore, we can simplify the formula to $\mathit{Post}(\mathsf{z}, \mathsf{z} * 2) \leftrightarrow \mathit{Post}(\mathsf{z}, \mathsf{z} * 2)$ which is valid.

Finally, we have a look at the normal completion condition of the instantiation for AS A. The corresponding formula (with premise "true" due to the instantiations of all abrupt completion preconditions as false) to prove is

$$\mathit{normalCompletionFor}(\mathit{specs}, p)[\mathit{subst}_{\mathit{funcAndPredSymbols}}][\mathit{subst}_{\mathit{locSpecVars}}] = \text{true} \rightarrow$$
$$[\texttt{\_normal=}\textbf{false}\texttt{;}\ \textbf{exec}\ \texttt{\{x=z;z*=2;}\ \textbf{break}\ l\texttt{;\}}\ \textbf{ccatch}\ \texttt{(\textbackslash}\textbf{Break}\ l\texttt{)}\ \texttt{\{\_normal=}\textbf{true}\texttt{;\}}]\ (\texttt{\_normal} \doteq \text{TRUE} \wedge \text{true})$$

As the labeled **break** statement is always reached and the trivial postcondition true holds, this formula is valid, too. We conclude that the concrete program fragment p^0 is a legal instantiation of the abstract program fragment $\mathscr{F}$. Note that it would have been possible to choose the APEs A and B differently in `x=z; z*=2; i=2k;` when trying to match the concrete to the abstract program, i.e., $A :=$ `x=z;` and $B :=$ `z*=2;i=2k;`. Then, however, it is not possible to prove disjointness of *frameB* and *footprintA*, which become $\mathit{frameB} = \{\mathring{z}, \mathring{i}\}$ and $\mathit{footprintA} = \{\mathring{z}\}$. Mapping a concrete to an abstract program (without further hints) can be expensive; There might be many possibilities which have to be checked, of which only some are actual instances, if any. $\Diamond$

Next, we define syntax and semantics of *abstract updates* representing underspecified state changes caused by APEs.

4.2.3 Syntax and Semantics of Abstract Updates

Updates are essential in JavaDL. They are *modal operators*, but in contrast to (diamond or box) modalities, they always terminate, and the expressions occurring in right-hand sides of updates never have side effects. The principle of Symbolic Execution in JavaDL is to represent state changes as updates and different control flows as separate branches in the proof tree. Thus, all side effects with the exception of "pure" state changes in form of updates are eliminated. The latter are ultimately applied to the post conditions, giving rise to first-order problems. As we cannot transform abstract programs containing APEs into *concrete* updates, we introduce a new syntactic category called *abstract updates*.

An abstract update has the form $\mathscr{U}_{\mathsf{P}}(\mathit{assignables} :\approx \mathit{accessibles})$. We on purpose use the same syntax as in the shorthand notation for APEs without abrupt completion. The basic idea is indeed to transform an APE $\mathsf{P}(\mathit{assignables} :\approx \mathit{accessibles})$ to an abstract update

$\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)$ with the same semantics: The update writes *at most* to the locations specified in *assignables*, while being parametric in *accessibles*. Similar to APEs, abstract updates have an identifier symbol, in the present example "$\mathcal{U}_{\mathsf{P}}$". We generally index abstract updates with the APEs for which they were introduced to increase understandability. However, we do not enforce a formal semantic connection between APEs and "their" abstract updates. The only connection enforced is relevant for the completeness of the AE calculus described in Sect. 4.3: We always use the same *name* for abstract updates generated for APEs with the same identifier symbol.

The following definition adds a new category of *abstract update symbols* to signatures. An abstract update symbol is an operator with a name (such as "$\mathcal{U}_{\mathsf{P}}$"), a list of parameters (its assignable locations), and an arity. Formally, the parameters are a list of location set terms. Those are *part of the operator itself*: Replacing a parameter, for instance by removing a "has-to" annotation, yields a *different operator*. *Abstract updates* are created from applying abstract update *symbols* to a list of terms (the right-hand sides, or "accessibles"). The length of the list has to match the arity of the symbol. Abstract updates can be used in the construction of sequential and parallel updates and update applications.

Definition 4.6 (Abstract Updates). We extend JavaDL signatures Σ by a set $\mathrm{Upd}^{\mathscr{A}}$ of (parametric) *abstract update symbols* $\mathcal{U}_{\mathsf{P}}(assignables)$, where $\mathcal{U}_{\mathsf{P}}$ is the identifier of the abstract update symbol, and its parameter list $assignables \subseteq (\mathrm{Trm}_{LocSet})^n$ is an n-tuple consisting of *LocSet* terms. Each abstract update symbol with the same identifier has (1) the same number n of assignable locations, and (2) the same *arity* m. To the set Upd of updates we add, for $\mathcal{U}_{\mathsf{P}}(assignables) \in \mathrm{Upd}^{\mathscr{A}}$, expressions $\mathcal{U}_{\mathsf{P}}(assignables :\approx accessibles)$ ("abstract updates"), which may occur in compound update constructions. The right-hand side $accessibles \subseteq (\mathrm{Trm}_{Any})^m$ is an m-tuple of argument terms, where $m \in \mathbb{N}$ is the arity of the abstract update symbol. ◊

To define the semantics of abstract updates, we extend the interpretation function I of JavaDL Kripke structures such that $I(\mathcal{U}_{\mathsf{P}}(assignables))$ returns a function $(\mathscr{D})^m \to \mathscr{S} \to \mathscr{S}$ which, depending on the values of the right-hand side of an abstract update, returns a state transformer. We then extend the valuation function of dynamic logic accordingly. The interpretation of an abstract update symbol has to respect its "frame" (i.e., *assignables*). Furthermore, we have to ensure that the interpretation of abstract updates *with the same identifier* is equivalent "modulo frame changes". For instance, the abstract update $\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{x}} :\approx accessibles)$ should have the same effect on x that $\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{y}} :\approx accessibles)$ has on y, such that it holds that $\mathsf{x} \doteq \mathsf{y} \to \{\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{x}} :\approx accessibles)\}\mathsf{x} \doteq \{\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{y}} :\approx accessibles)\}\mathsf{y}$. We need the premise because the left-hand sides in the abstract updates are not declared as "has-to": They do not have to be written, in which case the variables have to be equal in the pre-state for the equality to hold. For has-to locations, we would not need the premise, which gives rise to the constraint on their semantics: For whatever value the locations x and y attain in the pre-state, their values are equal after transformation by the updates.

Definition 4.7 (Semantics of Abstract Updates)**.** An interpretation function I of a JavaDL Kripke structure $(\mathcal{D}, \delta, I, \mathcal{S}, \varrho)$ assigns to an abstract update symbol $\mathcal{U}_{\mathsf{P}}(\mathit{assignables}) \in \mathrm{Upd}^{\mathcal{A}}$ with arity m a function $(\mathcal{D})^m \to \mathcal{S} \to \mathcal{S}$, such that

(1) *Frame Condition:* Let $\mathit{accessibles} \in (\mathcal{D})^m$ and $\sigma \in \mathcal{S}$. For all locations $\mathit{loc} \in \mathcal{D}^{\mathit{LocSet}}$, it holds that either $\mathit{loc} \in \mathit{val}(K, \sigma | \mathit{assignables})$, or

$$\sigma(\mathit{loc}) = I(\mathcal{U}_{\mathsf{P}}(\mathit{assignables}))(\mathit{accessibles})(\sigma)(\mathit{loc}).$$

(2) *State Transformers for Same Identifier Are Isomorphic:* Let, for any $i = 1, \dots, n$, be $\mathcal{U} = \mathcal{U}_{\mathsf{P}}(s_1, \dots, s_i, \dots, s_n) \in \mathrm{Upd}^{\mathcal{A}}$, $\mathit{accessibles} \in (\mathcal{D})^m$ and $\sigma \in \mathcal{S}$. For all location set terms s_i' representing the same number of concrete locations as s_i (i.e., $|\mathit{val}(K, \sigma | s_i)| = |\mathit{val}(K, \sigma | {s_i}')|$), there has to be a bijective mapping ι between $\mathit{val}(K, \sigma | s_i)$ and $\mathit{val}\big(K, \sigma | s_i'\big)$, such that for all $\mathit{loc} \in \mathit{val}(K, \sigma | s_i)$, it holds that

$$I(\mathcal{U})(\mathit{accessibles})(\sigma)(\mathit{loc}) = I(\mathcal{U}')(\mathit{accessibles})(\sigma')(\iota(\mathit{loc})),$$

where $\mathcal{U}' := \mathcal{U}_{\mathsf{P}}(s_1, \dots, s_i', \dots, s_n)$ and $\sigma' := \sigma[\iota(\mathit{loc}) \mapsto \sigma(\mathit{loc})]$.

(3) *Has-To Condition:* For $\mathcal{U} = \mathcal{U}_{\mathsf{P}}(s_1, \dots, s_i^!, \dots, s_n)$, the requirement of Item (2) has to hold for has-to locations $s_i' = (s_i'')^!$ and $\sigma' := \sigma$. $\Diamond$

The mapping ι in Item (2) of Def. 4.7 is required because a single element of the *assignables* of an abstract update symbol can be an *abstract* location set and therefore represent many concrete locations. The definition requires that state transformers created for the two abstract updates with the same identifier and equal *accessibles* arguments transform a pre-state σ, where a location *loc* has the same value as its corresponding location $\iota(\mathit{loc})$, to a state where they *still* have the same value (though potentially different from the value in σ). If *loc* is a “has-to” location, the value in the resulting state will be equal independent of the value in the pre-state.

We illustrate the semantics of abstract updates along an example.

Example 4.5 (Semantics of Abstract Update Symbols)**.** Let $\mathcal{U}_{\mathsf{P}}$ be an abstract update name and $\mathcal{U} := \mathcal{U}_{\mathsf{P}}(\mathit{locs})$ an abstract update symbol. Its single parameter is an abstract location set *locs*. Let locs' be a different abstract location set; the abstract update symbol $\mathcal{U}' := \mathcal{U}_{\mathsf{P}}(\mathit{locs}')$ has the same *name* as $\mathcal{U}$, but a different parameter, and is thus a different symbol. However, their semantics is coupled by Def. 4.7. Assume that I interprets *locs* as $\{\underline{\mathtt{x}}, \underline{\mathtt{y}}\}$, and locs' as $\{\underline{\mathtt{y}}, \underline{\mathtt{z}}\}$. Due to Item (1) in Def. 4.7, the result of $I(\mathcal{U})(\mathit{accessibles})(\sigma)$

may only differ from σ in the values of variables x and y, for any tuple *accessibles* of accessible locations. In the case of $\mathcal{U}'$, the same holds for y and z. Due to Item (2), interpretations of $\mathcal{U}$ and $\mathcal{U}'$ have to transform, given the same parameters, an input state σ to output states coinciding in the value of y, as well as in the values of x in one and z in the other state. Formally, the only bijective mapping ι between the interpretations of *locs* and *locs*$'$ associates y itself and x with z (and vice versa). Then, it must hold that

$$\begin{aligned} I(\mathcal{U})(\mathit{accessibles})(\sigma)(\mathsf{y}) &= I(\mathcal{U}')(\mathit{accessibles})(\sigma)(\mathsf{y}) \qquad \text{and} \\ I(\mathcal{U})(\mathit{accessibles})(\sigma)(\mathsf{x}) &= I(\mathcal{U}')(\mathit{accessibles})(\sigma[\mathsf{z} \mapsto \sigma(\mathsf{x})])(\mathsf{z}) \end{aligned}$$

Assume for simplicity that $\mathcal{U}$ and $\mathcal{U}'$ have arity 1 (by Def. 4.6, they have the same arity), and that $I(\mathcal{U})$ assigns to x the double value of the input, i.e., $I(\mathcal{U})(21)(\sigma)(\mathsf{x}) = 42$. The same results for $\mathcal{U}'$ and z: $I(\mathcal{U}')(21)(\sigma[\mathsf{z} \mapsto \sigma(\mathsf{x})])(\mathsf{x}) = 42$. Def. 4.7 also allows interpretations of $\mathcal{U}$, $\mathcal{U}'$ not changing the value of x, i.e., $I(\mathcal{U})(21)(\sigma)(\mathsf{x}) = \sigma(x)$. For this reason, we have to update the input state for $\mathcal{U}'$ according to ι:

$$I(\mathcal{U}')(21)(\sigma[\mathsf{z} \mapsto \sigma(\mathsf{x})])(\mathsf{z}) = \sigma[\mathsf{z} \mapsto \sigma(\mathsf{x})](\mathsf{z}) = \sigma(\mathsf{x}) = I(\mathcal{U})(21)(\sigma)(\mathsf{x}).$$

If *locs* in $\mathcal{U}$, $\mathcal{U}'$ was designated as "has-to" (this would yield two *different* abstract update symbols), the state update "$[\mathsf{z} \mapsto \sigma(\mathsf{x})]$" is removed, i.e., the interpretation of the updates is not allowed to leave the locations represented by *locs* unassigned:

$$I(\mathcal{U}_{\mathsf{P}'}(\mathring{\mathsf{z}}^!, \mathring{\mathsf{y}}^!))(21)(\sigma)(\mathsf{z}) \overset{\text{has to}}{=} I(\mathcal{U}_{\mathsf{P}'}(\mathring{\mathsf{x}}^!, \mathring{\mathsf{y}}^!))(21)(\sigma)(\mathsf{x}). \qquad \Diamond$$

We extend the evaluation function $\mathit{val}(K, \sigma, \beta|\cdot)$ for abstract updates.

Definition 4.8 (Valuation of Abstract Updates)**.** We extend the JavaDL valuation function $\mathit{val}(K, \sigma, \beta|\cdot)$ of Sect. 2.2.2 as follows, for $\mathcal{U}_{\mathsf{P}}(\mathit{assignables}) \in \mathrm{Upd}^{\mathcal{A}}$ with arity m:

$$\begin{aligned} &\mathit{val}(K, \sigma, \beta|\mathcal{U}_{\mathsf{P}}(\mathit{assignables} :\approx t_1, \ldots, t_m)) = \\ &\quad I(\mathcal{U}_{\mathsf{P}}(\mathit{assignables}))(\mathit{val}(K, \sigma, \beta|t_1), \ldots, \mathit{val}(K, \sigma, \beta|t_m)) \end{aligned} \qquad \Diamond$$

So far, we looked into the fine-grained building blocks of our logic: Abstract program elements, program fragments and updates. We conclude the syntax and semantics section by assembling those to abstract sequents and SESs.

4.2.4 Syntax and Semantics of Abstract Sequents and SESs

We first introduce a generalization of (JavaDL validity calculus) sequents to *abstract* sequents via an extension of the valuation function for formulas containing abstract program fragments inside modalities, and subsequently generalize SESs to *abstract* SESs.

Abstract JavaDL formulas differ from JavaDL formulas (cf. Sect. 2.2.1) by modalities $[\mathscr{F}]$ and $\langle\mathscr{F}\rangle$ *with abstract program fragments*. All APEs *ape* in a diamond modality $\langle\mathscr{F}\rangle$ have to satisfy $term = \text{TOTAL}$, while those in a box modality $[\mathscr{F}]$ may not terminate ($term = \text{PARTIAL}$). To define the semantics of abstract JavaDL, we have to give meaning to formulas $[\mathscr{F}]\varphi$ and $\langle\mathscr{F}\rangle\varphi$. As AE has the goal to prove *universal* second-order program properties, we quantify over all legal instances of the abstract program fragment.

Definition 4.9 (Semantics of Abstract JavaDL Formulas and Sequents)**.** Let $\mathscr{F}$ be an abstract program fragment. The semantics of the *Abstract JavaDL Formula* $[\mathscr{F}]\varphi$ is defined such that $val(K,\sigma|[\mathscr{F}]\varphi) = tt$ if, and only if, for all concrete program fragments $p \in [\![\mathscr{F}]\!]$ it holds that $val(K,\sigma|[p]\varphi) = tt$ (similarly for $\langle\mathscr{F}\rangle\varphi$)). For formulas with multiple abstract program fragments, we require that the individual substitutions for APEs, specification variables and constraints can be combined without conflicts (i.e., two APEs occurrences in different modalities within the same formula, for instance, have to be behaviorally isomorphic if they have the same identifier). Apart from that, the semantics is defined as for concrete JavaDL formulas. The semantics of an abstract JavaDL *sequent* $\Gamma \vdash \Delta$ is, as usual, defined as the semantics of the formula $\bigwedge\Gamma \rightarrow \bigvee\Delta$. ◊

Remark 4.8 (Expressiveness of Abstract Execution). To assess the expressiveness of AE, it helps to regard a formula containing abstract program fragments as a formula of JavaDL$^{\text{II}}$ with free second-order variable symbols ranging over statements and expressions, and an implicit block of premises constraining their domain.

Let $\varphi_{\text{JavaDL}}(ape_1,\ldots,ape_n)$ be an abstract JavaDL formula with occurrences of n APEs ape_1 to ape_n. Its semantics corresponds to that of a JavaDL$^{\text{II}}$ formula

$$\begin{aligned}&\forall^{\text{II}} v_1 : \mathit{Stmt/Expr}; \ldots; \forall^{\text{II}} v_n : \mathit{Stmt/Expr};\\ &\qquad\Big(\bigwedge_{i_1,\ldots,i_k:(\star)} \mathit{consistent}(v_{i_1},\ldots,v_{i_k}) \wedge \bigwedge_{i=1,\ldots,n} \mathit{represents}(ape_i, v_i) \rightarrow \varphi_{\text{JavaDL}}(v_1,\ldots,v_n)\Big)\end{aligned} \tag{4.1}$$

where $(\star)$: "$ape_{i_1},\ldots,ape_{i_k}$ have the same identifier symbol", and *consistent* is a syntactic representation of requirement (5) in Def. 4.5. We assume that when instantiating a second-order statement or expression variable, specification variables connected to the corresponding APEs are instantiated accordingly. For instance, a dynamic frame variable is instantiated with the locations that are actually written, and the precondition for exceptional completion with a formula describing the actual (necessary and sufficient) condition for abrupt completion due to an exception. The problem of instance checking for an abstract formula boils down to showing the validity of the second-order-quantifier-free core of Eq. (4.1) for a particular assignment of concrete program elements to $v_1, \ldots, v_n$.

Consequently, it is impossible to prove validity of a statement like "for every program with loops, there exists an equivalent version using only tail-recursive methods": First, we cannot express *existential quantification*, and secondly, we cannot express the property "without loops", which regards the internal structure of different programs with (for us) *indistinguishable behavior*. The *represents* expression used in Eq. (4.1) can only distinguish programs with *different external behavior* and is oblivious of internal structure.

What theoretically can be expressed, however, is the equivalence of all instantiations of two abstract programs where one contains a loop and one a tail-recursive method, both with abstract guards and bodies. This would correspond to the property "each loop can be converted into a behaviorally equivalent tail-recursive method, and vice versa", and thus get quite near the original property. Generally, the expressive power of our AE framework increases (or decreases) with the strength of *represents*. We would like to point out that, since we can impose arbitrary JavaDL postconditions for APEs and encode precise constraints on abstract specification variables, the class of *universal* properties that are not concerned about internal structure—at least not beyond a fixed degree of nesting—which we can express with AE is quite big. $\Diamond$

Generalizing the notion of a Symbolic Execution State to AE is straightforward: It suffices to use abstract instead of concrete program fragments as program counters. Def. 4.10 below extends Def. 3.1 accordingly.

Definition 4.10 (Abstract Symbolic Execution State)**.** An *Abstract* Symbolic Execution State is a triple $(C, \mathcal{U}, \mathcal{F})$ of (1) a path condition, formalized as a set of closed formulas $C \in 2^{\mathrm{Fml}}$, (2) a symbolic store, formalized as an update $\mathcal{U} \in \mathrm{Upd}$, and (3) a program counter, formally a legal *abstract* (Java) program fragment (cf. Def. 4.2) $\mathcal{F}$. We write $\mathbb{S}_{SE}^{\mathcal{A}}$ for the set of all abstract SESs. $\Diamond$

As for "concrete" SESs, the semantics of *abstract* SESs is based on a *concretization function* ultimately mapping symbolic to concrete states. The concretization function for abstract SESs takes an additional argument as input: A concrete program fragment. Then, the extension is straightforward. If the given program fragment is represented by the abstract program counter of the SES, the concretization for this concrete program fragment is part of the semantics; otherwise, the result is the empty set. The semantics of the abstract SES is obtained by building the union over all concrete program fragments.

Definition 4.11 (Concretization and Semantics of Abstract SESs). The *K-indexed abstract concretization function* $concr_K^{\mathscr{A}} : \mathbb{S}_{SE}^{\mathscr{A}} \times \mathscr{S} \times Stmt \to 2^{\mathscr{S}}$ maps an abstract SES $(C, \mathscr{U}, \mathscr{F})$, a concrete state $\sigma \in \mathscr{S}$ and concrete program element $p : Stmt$ (1) to the empty set $\emptyset$ if $p \notin [\![\mathscr{F}]\!]$, or (2) to the set $concr_K(C, \mathscr{U}, p)$ otherwise. The *abstract concretization function* $concr^{\mathscr{A}}$ is defined as $concr^{\mathscr{A}}(s, \sigma, p) := \bigcup_K (s, \sigma, p)$. The full semantics $[\![s]\!]$ of an abstract SES $s \in \mathbb{S}_{SE}^{\mathscr{A}}$ is defined as $[\![s]\!] := \bigcup_{\sigma \in \mathscr{S}} \bigcup_{p \in Stmt} concr^{\mathscr{A}}(s, \sigma, p)$. $\Diamond$

4.3 Abstract Execution Calculus

The term *Abstract Execution* can be seen as a short form for "symbolic *execution* of *abstract* programs". In the first part of this section, Sect. 4.3.1, we therefore present the heart of AE, that is, the symbolic execution calculus rules for abstract statements and expressions. These rules use *abstract updates* to describe state changes caused by APEs, depending on dynamic frame specification variables. In the last part (Sect. 4.3.2), we introduce dedicated calculus rules for simplifying terms and formulas with abstract updates. Furthermore, we describe calculus rules for dealing with the extensions to the *LocSet* theory which we introduced in the previous section.

4.3.1 Symbolic Execution Rules for APEs

Symbolic Execution rules for Abstract Execution are based on three principles:

(1) Second-order state changes based on *abstract updates*,
(2) creation of symbols "*dependently fresh*" for APE identifier symbols: They are created fresh on the *first* occurrence of an APE with an identifier symbol P in a proof context, but are *re-used* if, and only if, an APE with identifier symbol P re-occurs,
(3) creation of separate SE branches for all reasons of abrupt completion of an APE.

The whole AE rule for ASs would fill a page when written without abbreviations. To understand our rules, however, it suffices to understand these principles (and to have an intuition of the semantics of ASs and AExps, as detailed in the previous section). Abstract updates have been discussed already in Sect. 4.2. They relate to concrete updates in the same way that abstract programs relate to concrete ones: An abstract update represents a bunch of concrete state changes at once. Similarly to concrete updates, they always terminate, and the expressions occurring in abstract updates never have side effects. Based on principle (1), we can already construct a *sound AE rule*:

$$\textsf{abstractStatementSimple}\quad \frac{\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\}(\mathit{normalPost} \rightarrow [\pi\ \ \omega]\varphi), \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \texttt{\textbf{\textbackslash abstract\_statement}}\ \texttt{P;}\ \ \omega]\varphi, \Delta}\ (*)$$

where $(*)$ means that the rule only matches to ASs whose specification excludes abrupt completion. The term lists *frame* and *footprint* are created from the frame and footprint definition of the APE. All frame locations designated with **\hasTo**, e.g., **\hasTo**(x), are transformed into the corresponding logical representation $\mathring{\mathtt{x}}^{!}$. We continue speaking of "**\hasTo** locations" in these cases. The expression *value*(*footprint*) is an abbreviation for a list $\mathit{value}(\mathit{fp}_1), \ldots, \mathit{value}(\mathit{fp}_m)$ where the fp_i are the elements of the footprint of the APE. Furthermore, *normalPost* is the postcondition for normal completion. The abstract update symbol $\mathcal{U}_{\mathsf{P}}(\mathit{frame})$ itself is introduced fresh, but is *re-used* every time an AS with the identifier symbol P is processed again by the rule. This incorporates principle (2) into the rule; we call this process "*dependently* fresh" introduction of logic symbols.

The missing puzzle piece is that we would like to drop condition $(*)$, and also execute ASs which may complete abruptly. So far, abrupt completion has to be excluded since it is not modeled in the premise of the rule. We generalize the rule by considering principle (3). In a first step, we admit abrupt completion due to an *exception* only, i.e., instantiations of the APE may still not return, break, etc. To analyze the effect of abrupt completion of the APE due to a thrown exception, we add a separate SE branch for the case where that exception has been thrown. A straightforward way to accomplish this is to add a conditional Java statement in the modality at the place of the APE, which, depending on the value of a *symbolic* guard, throws a *symbolic* exception object:

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint})\}((\boxed{\texttt{throwsExc} \not\doteq \mathrm{TRUE} \rightarrow} \mathit{normalPost}) \rightarrow \\ \qquad [\pi\ \ \boxed{\texttt{\textbf{if} (throwsExc) \textbf{throw} exc;}}\ \ \omega]\varphi), \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \texttt{\textbf{\textbackslash abstract\_statement}}\ \texttt{P;}\ \ \omega]\varphi, \Delta}$$

This mimics the effects of this type of abrupt completion, especially on the control flow. On the other hand, it constitutes an overapproximation, since the program variables throwsExc and exc are introduced freshly, with, so far, no additional information connected to them. The postcondition *normalPost* has to be guarded by the negation of `throwsExc`, since it should not hold for abrupt completion.

In this state, principle (2) is not respected, since the values of `throwsExc` and `exc` are not coupled for occurrences of different APEs with the same identifier symbol. To address this shortcoming, we initialize both variables with the values of two terms $\mathit{throwsExc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))$ and $\mathit{exc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))$ that are created *dependently fresh* for the APEs with identifier symbol P, as indicated by the superscript P in the names of the functions. The types of the functions are $\mathit{throwsExc}^{\mathsf{P}} : \mathit{Any} \to \mathit{boolean}$ and $\mathit{exc}^{\mathsf{P}} : \mathit{Any} \to$ Exception. Note that these terms depend on the value of the APE's footprint in the current state, since the same program element may, or may not, throw an exception when it is executed in different states. Additionally, we do not want the thrown exception to be **`null`**. This is for practical reasons: Writing "**`throw null`**`;`" in a Java program is absolutely possible, but equivalent to writing "**`throw new`** `NullPointerException();`". Therefore, **`null`** is not a value that can *actually* be thrown. Treating this special case, however, leads to significant overhead in KeY proofs, which we save by the additional assumption. The resulting, refined rules is:

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}\{\mathtt{throwsExc} := \mathit{throwsExc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\}\\ \qquad \{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\}\\ \qquad\quad \{\mathtt{exc} := \mathit{exc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\}\\ \qquad\qquad ((\mathtt{throwsExc} \doteq \mathrm{TRUE} \to \mathtt{exc} \neq \mathbf{null}) \land\\ \qquad\qquad\ (\mathtt{throwsExc} \not\doteq \mathrm{TRUE} \to \mathit{normalPost}) \to\\ \qquad\qquad\qquad [\pi\ \mathbf{if}\ \mathtt{(throwsExc)}\ \mathbf{throw}\ \mathtt{exc;}\ \omega]\varphi), \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathbf{\backslash abstract\_statement}\ \mathtt{P;}\ \omega]\varphi, \Delta}$$

This rule is unsound: It does not take into account the specified pre- and postconditions for completion due to an exception of the APE, and therefore does not conform to the semantics of APEs. We add these constituents in a final step. Furthermore, in preparation for the subsequent addition of other reasons of abrupt completion, we introduce a fresh boolean program variable `normal` which evaluates to TRUE iff none of the guards for abrupt completion evaluates to TRUE.[4]

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}\{\texttt{throwsExc} := \mathit{throwsExc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\} \\ \quad \big((\texttt{normal} \doteq \mathrm{TRUE} \leftrightarrow \texttt{throwsExc} \neq \mathrm{TRUE}) \land \\ \quad (\texttt{throwsExc} \doteq \mathrm{TRUE} \leftrightarrow \mathit{pre}(\mathit{excSpec}))\big) \rightarrow \\ \quad \{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\} \\ \quad\quad \{\texttt{exc} := \mathit{exc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\} \\ \quad\quad\quad ((\texttt{throwsExc} \doteq \mathrm{TRUE} \rightarrow \texttt{exc} \neq \textbf{null} \land \mathit{post}(\mathit{excSpec})) \land \\ \quad\quad\quad (\texttt{normal} \doteq \mathrm{TRUE} \rightarrow \mathit{normalPost}) \rightarrow \\ \quad\quad\quad\quad [\pi\ \textbf{if}\ \texttt{(throwsExc)}\ \textbf{throw}\ \texttt{exc;}\ \omega]\varphi), \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \textbf{\textbackslash abstract\_statement}\ \texttt{P;}\ \omega]\varphi, \Delta}$$

This rule is sound and applies to ASs whose specification allows them to complete normally *or* due to a thrown exception. Adding further reasons for abrupt completion only enlarges the textual representation of the rule, but not its conceptual complexity. The complete AE rule for SE of an AS is shown in Fig. 4.1. It contains abbreviations to keep it readable. The function *mutex* is defined such that *at most* one of its (boolean) arguments may evaluate to TRUE. The labels lb_{b_1} to lb_{b_n} are all (distinct) loop or block labels declared in the prefix π, while the labels lb_{c_1} to lb_{c_m} are all loop labels only.

The update $\mathcal{U}_{init}$ initializes all symbolic boolean flags, like `throwsExc` and `returns`, which we have seen before, to the terms created based on functions chosen "dependently fresh" for the AS, which depend on the current value of the footprint. The terms for labeled **`break`**s and **`continue`**s additionally receive the label as an argument (alternatively, we could have introduced different function symbols for each label).

$$\begin{aligned}\mathcal{U}_{init} := \ & \texttt{throwsExc} := \mathit{throwsExc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\ \| \\ & \texttt{returnsVal} := \mathit{returnsVal}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\ \| \\ & \texttt{returns} := \mathit{returns}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\ \| \\ & \texttt{breaks} := \mathit{breaks}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\ \| \\ & \texttt{continues} := \mathit{continues}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\ \| \\ & \texttt{breaks\_}\mathit{lb}_{b_1} := \mathit{breaksLb}^{\mathsf{P}}(\mathit{lb}_{b_1}, \mathit{value}(\mathit{footprint}))\ \| \cdots \|\end{aligned}$$

[4] Java programs, and thus also APEs, complete normally iff they do not complete abruptly; consequently, we do not have a special precondition (such as $\mathit{throwsExc}^{\mathsf{P}}$ for exceptional completion) for *normal* completion.

abstractStatement

$$
\frac{
\begin{array}{l}
\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{init}\} \\
\quad \big(\mathit{mutex}(\texttt{throwsExc}, \texttt{returnsVal}, \texttt{returns}, \texttt{breaks}, \texttt{continues}, \\
\qquad \texttt{breaks\_}\mathit{lb}_{b_1}, \ldots, \texttt{breaks\_}\mathit{lb}_{b_n}, \\
\qquad \texttt{continues\_}\mathit{lb}_{c_1}, \ldots, \texttt{continues\_}\mathit{lb}_{c_m}) \land \\
\quad (\texttt{normal} \doteq \mathrm{TRUE} \leftrightarrow \mathit{notAbruptly}) \land \\
\quad \mathit{behavioralPreconds}\big) \rightarrow \\
\quad \{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\} \\
\quad \{\texttt{exc} := \mathit{exc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint})) \,||\, \texttt{res} := \mathit{res}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\} \\
\quad \big(\mathit{behavioralPostconds} \rightarrow \\
\quad [\pi\ \texttt{\textbf{if} (throwsExc) \textbf{throw} exc;} \\
\qquad \texttt{\textbf{if} (returnsVal) \textbf{return} res;} \\
\qquad \texttt{\textbf{if} (returns) \textbf{return};} \\
\qquad \texttt{\textbf{if} (breaks) \textbf{break};} \\
\qquad \texttt{\textbf{if} (continue) \textbf{continue};} \\
\qquad \texttt{\textbf{if} (breaks\_}\mathit{lb}_{b_1}\texttt{) \textbf{break} } \mathit{lb}_{b_1}\texttt{;} \cdots \\
\qquad \texttt{\textbf{if} (breaks\_}\mathit{lb}_{b_n}\texttt{) \textbf{break} } \mathit{lb}_{b_n}\texttt{;} \\
\qquad \texttt{\textbf{if} (continues\_}\mathit{lb}_{c_1}\texttt{) \textbf{continue} } \mathit{lb}_{c_1}\texttt{;} \cdots \\
\qquad \texttt{\textbf{if} (continues\_}\mathit{lb}_{c_m}\texttt{) \textbf{continue} } \mathit{lb}_{c_m}\texttt{;}\ \omega]\varphi\big), \Delta
\end{array}
}{
\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{\textbf{\textbackslash abstract\_statement} P;}\ \omega]\varphi, \Delta
}
$$

Figure 4.1: The Abstract Execution Rule for Abstract Statements. (Abbreviations and label symbols are explained in the text)

$$\begin{aligned}
&\texttt{breaks_}lb_{b_n} := breaksLb^{\mathsf{P}}(lb_{b_n}, value(footprint)) \mid\mid \\
&\texttt{continues_}lb_{c_1} := continuesLb^{\mathsf{P}}(lb_{c_1}, value(footprint)) \mid\mid \cdots \mid\mid \\
&\texttt{continues_}lb_{c_m} := continuesLb^{\mathsf{P}}(lb_{c_m}, value(footprint))
\end{aligned}$$

Where before, we specified that the AS terminates normally iff it does not throw an exception, we now have to consider *all* reasons for abrupt completion, which occur in the abbreviation *notAbruptly* defined as follows:

$$\begin{aligned}
notAbruptly := {} & \neg\texttt{throwsExc} \doteq \mathrm{TRUE} \land \neg\texttt{returnsVal} \doteq \mathrm{TRUE} \land \\
& \neg\texttt{returns} \doteq \mathrm{TRUE} \land \neg\texttt{breaks} \doteq \mathrm{TRUE} \land \neg\texttt{continues} \doteq \mathrm{TRUE} \land \\
& \neg\texttt{breaks_}lb_{b_1} \doteq \mathrm{TRUE} \land \cdots \land \neg\texttt{breaks_}lb_{b_n} \doteq \mathrm{TRUE} \land \\
& \neg\texttt{continues_}lb_{c_1} \doteq \mathrm{TRUE} \land \cdots \land \neg\texttt{continues_}lb_{c_m} \doteq \mathrm{TRUE}
\end{aligned}$$

The formula *behavioralPreconds* binds the values of the boolean flags for all reasons of abrupt completion to the preconditions in the *specs* element of the AS. Since the specifications for labeled **`break`**s and **`continue`**s are parametric in the label, the corresponding formulas in the following definition are also passed the label as a parameter:

$$\begin{aligned}
behavioralPreconds := {} & (\texttt{throwsExc} \doteq \mathrm{TRUE} \leftrightarrow pre(excSpec)) \land \\
& (\texttt{returnsVal} \doteq \mathrm{TRUE} \leftrightarrow pre(returnsValSpec)) \land \\
& (\texttt{returns} \doteq \mathrm{TRUE} \leftrightarrow pre(returnsSpec)) \land \\
& (\texttt{breaks} \doteq \mathrm{TRUE} \leftrightarrow pre(breaksSpec)) \land \\
& (\texttt{continues} \doteq \mathrm{TRUE} \leftrightarrow pre(continuesSpec)) \land \\
& (\texttt{breaks_}lb_{b_1} \doteq \mathrm{TRUE} \leftrightarrow pre(breaksSpecLbl(lb_{b_1}))) \land \cdots \land \\
& (\texttt{breaks_}lb_{b_n} \doteq \mathrm{TRUE} \leftrightarrow pre(breaksSpecLbl(lb_{b_n}))) \land \\
& (\texttt{continues_}lb_{c_1} \doteq \mathrm{TRUE} \leftrightarrow pre(continuesSpecLbl(lb_{c_1})) \land \cdots \land \\
& (\texttt{continues_}lb_{c_m} \doteq \mathrm{TRUE} \leftrightarrow pre(continuesSpecLbl(lb_{c_m})))
\end{aligned}$$

Finally, abbreviation *behavioralPostconds* adds the assumptions about all postconditions:

$$\begin{aligned}
behavioralPostconds := {} & (\texttt{normal} \doteq \mathrm{TRUE} \rightarrow post(normalSpec)) \land \\
& (\texttt{throwsExc} \doteq \mathrm{TRUE} \rightarrow post(excSpec) \land \texttt{exc} \not\doteq \texttt{null}) \land \\
& (\texttt{returnsVal} \doteq \mathrm{TRUE} \rightarrow post(returnsValSpec)) \land
\end{aligned}$$

$$
\begin{aligned}
&(\texttt{returns} \doteq \mathrm{TRUE} \rightarrow \mathit{post}(\mathit{returnsSpec}))\land\\
&(\texttt{breaks} \doteq \mathrm{TRUE} \rightarrow \mathit{post}(\mathit{breaksSpec}))\land\\
&(\texttt{continues} \doteq \mathrm{TRUE} \rightarrow \mathit{post}(\mathit{continuesSpec}))\land\\
&(\texttt{breaks\_}\mathit{lb}_{b_1} \doteq \mathrm{TRUE} \rightarrow \mathit{post}(\mathit{breaksSpecLbl}(\mathit{lb}_{b_1}))) \land \cdots \land\\
&(\texttt{breaks\_}\mathit{lb}_{b_n} \doteq \mathrm{TRUE} \rightarrow \mathit{post}(\mathit{breaksSpecLbl}(\mathit{lb}_{b_n})))\land\\
&(\texttt{continues\_}\mathit{lb}_{c_1} \doteq \mathrm{TRUE} \rightarrow \mathit{post}(\mathit{continuesSpecLbl}(\mathit{lb}_{c_1})) \land \cdots \land\\
&(\texttt{continues\_}\mathit{lb}_{c_m} \doteq \mathrm{TRUE} \rightarrow \mathit{post}(\mathit{continuesSpecLbl}(\mathit{lb}_{c_m})))
\end{aligned}
$$

Note that according to Def. 4.1, preconditions have to be mutually exclusive, which is why we can bind them to the boolean flags with equivalences "$\leftrightarrow$" in *behavioralPreconds*. This requirement does not, and usually will not, have to hold for postconditions, which is why we use normal implications "$\rightarrow$" in *behavioralPostconds*.

Remark 4.9 (Omission of Specification Cases and Context-Sensitivity). In practice, it is not always desirable to explicitly specify all abrupt completion cases (cf. also Remark 4.5). This is for two reasons: First, sometimes the default behavior, i.e., that an AExp can, for instance, non-deterministically "decide" to throw an exception, is completely sufficient, and explicitly binding the precondition for exceptional completion to an expression `throwsExcP(\value(footprintP))` seems to be superfluous since we never use `throwsExcP` at other places, and second, in certain contexts some reasons for abrupt completion are just impossible. For instance, within a void method, a return *of a value* may never happen, and always adding the specification line

```
//@ return_val_behavior requires false;
```

leads to "syntactic noise". We address this by two measures. All pre- or postconditions for behavior specifications of an APE may be omitted. Relevant conjuncts of *behavioralPreconds* and/or *behavioralPostconds*, e.g., $(\texttt{throwsExc} \doteq \mathrm{TRUE} \leftrightarrow \mathit{pre}(\mathit{excSpec}))$, are then left out. Since $\mathcal{U}_{init}$ binds `throwsExc` to a term with a function created dependently fresh for the current APE which depends on the value of the footprint, this is sound, and still assures that two APEs with the same identifier behave equivalently in equivalent contexts. Furthermore, we specialize abstractStatement for different program contexts $\pi\ \omega$. For instance, if there is no loop scope opening in π, we omit every part of the rule that is related to unlabeled **`break`** and labeled or unlabeled **`continue`** statements. Theoretically, however, the single rule abstractStatement is sufficient for the AE of abstract statements, and all others can be derived from it. $\Diamond$

The rule for AExps, abstractExpression (depicted in Fig. 4.2), follows the same principles as abstractStatement. However, it is simpler, since expressions can only complete *abruptly* due to a thrown exception. What further distinguishes an expression from a statement is that it evaluates to a *value*.[5] The rule abstractExpression reflects this aspect as follows. The conclusion of the rule is a Java statement containing an AExp: "`v=`**`\abstract_expression`** T `e;`". Existing JavaDL rules establish this normal form for all "non-simple" expressions. If a complex expression occurs, e.g., as guard of an **`if`** statement, a new variable `v` is introduced and an assignment of the expression to `v` added before the **`if`**. In the rule, we initialize a variable `res`, which can be refined in the postcondition *normalPost* by constraining the value of the **`\result`** variable, with an expression $\mathit{res}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))$. Variable `res` is then assigned to `v` if the AExp does *not* complete abruptly due to a thrown exception; otherwise, we throw the exception.

$$
\text{abstractExpression}\quad
\frac{\begin{array}{l}
\Gamma \vdash \{\mathcal{U}\}\{\mathtt{throwsExc} := \mathit{throwsExc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))\} \\
\quad (\mathtt{throwsExc} \doteq \mathrm{TRUE} \leftrightarrow \mathit{pre}(\mathit{excSpec})) \rightarrow \\
\quad\quad \{\mathcal{U}_{\mathsf{e}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\} \\
\quad\quad\quad \{\mathtt{exc} := \mathit{exc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})) \parallel \mathtt{res} := \mathit{res}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))\} \\
\quad\quad\quad\quad \big((\mathtt{throwsExc} \doteq \mathrm{TRUE} \rightarrow \mathtt{exc} \not\doteq \mathtt{null} \land \mathit{post}(\mathit{excSpec})) \land \\
\quad\quad\quad\quad\; (\mathtt{throwsExc} \not\doteq \mathrm{TRUE} \rightarrow \mathit{normalPost}) \rightarrow \\
\quad\quad\quad\quad\quad [\pi\ \mathbf{if}\ \mathtt{(throwsExc)}\ \mathbf{throw}\ \mathtt{exc;} \\
\quad\quad\quad\quad\quad\quad \mathtt{v = res;}\ \omega]\varphi\big), \Delta
\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{v=}\mathbf{\backslash abstract\_expression}\ T\ \mathtt{e;}\ \omega]\varphi, \Delta}
$$

Figure 4.2: The Abstract Execution Rule for Abstract Expressions

Remark 4.10 (Abstract Expressions and Mode-Dependent Frames). The framework presented in the original AE paper [SH19a] did not have abstract expressions. Instead, an "abstract expression idiom" was used, where a fresh program variable of the type of the expression in conjunction with an AS setting that variable served as a substitute for a direct support of AExps. This has some practical disadvantages, like less understandable code, disruptive obligatory specifications that, e.g., forbid the AS to return (which an exception cannot), and the unsuitability for usage in loop guards. More interesting, however, is

[5] *Expression statements*, such as `i++` or calls to non-void methods, are in the intersection of statements and expressions. From the context of their appearance, it is always clear whether they represent a statement or expression, and which SE rule has to be applied.

the setting of the variable storing the evaluation result of the abstract expression. This variable was added as "**\hasTo**" to the frame of the AS in [SH19a]—which is, strictly speaking, unsound. In the case of abrupt completion of the expression, its result is not a value, but the exception. In this case, the result variable must not be assigned. It *would* be sound to add the result variable to the frame without the **\hasTo** specifier, which, however, is *incomplete*, since even in the case of normal completion, it would be unclear whether the statement set the variable or not. For this reason, the rule abstractExpression differentiates between the different modes of normal and abrupt completion and sets the result variable accordingly. The leads to *different frames* for the different completion modes. It could be interesting to add general support for mode-dependent frames to APEs, such that one can specify different frames (or footprints) for normal completion and the different kinds of abrupt completion. We abstain from doing so since this would increase the (already substantial) amount of specification lines in abstract program models: One would have to specify frames and footprints for all completion types. Of course, we could add defaults, which the user then had to know about. Furthermore, AE rules would get more complicated. Note that, since non-**\hasTo** frame elements constitute an "upper bound" to what can be written by an APE, it is possible to merge frames for different completion modes into one. When doing so, **\hasTo**s have to be removed if they do not apply for all behaviors. This approach is sound, albeit not precise. For the use cases to which we applied AE so far, the absence of mode-dependent frames was not a problem. Adding in the future, should it ever be required, still is an option. $\Diamond$

The most important criterion for our AE rules is that they are *sound*, i.e., we cannot prove falsity by using them. In addition, it would be desirable that they are *complete*, that is, the rules allow proving everything that is logically valid. This has to be understood modulo the inherent incompleteness owed to the presence of incomplete theories in JavaDL (cf. the notion of *relative completeness* in Prop. 2.2). One might however doubt that our AE rules are indeed complete, for two reasons:

(1) As mentioned before, we cannot prove certain second-order properties about programs, especially existential properties and those involving the *structure* of programs, as in "for all programs with exactly two nested loops, it has to hold that…". However, these problems cannot even be *expressed* in our framework, i.e., there is no meaningful syntax for them. To be complete, the rules have to be sufficiently precise to *allow for proving all properties that can be expressed* according to the definition of syntax and semantics of abstract programs, formulas, and sequents (Sect. 4.2).

(2) At first glance, *nontermination* of instances of an APE seems to constitute a completeness problem for the box modality—and even a soundness problem for diamond.

Consider rule abstractStatement with "$\langle \ldots \rangle$" instead of "$[\ldots]$". Soundness means that the validity of the conclusion follows from the validity of the premise. However, the premise contains no proof branches connected to termination, so the impression could arise that it is easier to prove validity for the premise than for the conclusion, which ranges over all legal instantiations of the AS. Yet, due to the definition of abstract JavaDL formulas, there *is no* non-terminating legal instantiation for an APE inside a diamond modality. For "$[\ldots]$", we might be concerned about completeness: That the validity of the conclusion implies the validity of the premise. If a program p inside a box modality formula $[\mathtt{p}]\varphi$ does not terminate, the whole formula is valid, regardless of φ. Nonetheless, since we can assume the validity of the conclusion *for all legal instantiations*, this comprises in particular all *terminating* legal instantiations. The key point is thus that we do not have to prove the premise assuming the validity of the conclusion for a single instantiation, but for all. Therefore, we do not need an SE branch like "`if (diverges) { while (true) {} }`".

Thms. 4.1 and 4.2 state the soundness of the AE rules for abstract statements and expressions, and Thms. 4.3 and 4.4 their completeness.

Theorem 4.1. *The AE rule* abstractStatement *(Fig. 4.1) is sound.*

Proof Sketch. We prove the validity of the conclusion of rule abstractStatement based on the assumption of the validity of the premise. The core insights used in the proof are:

(1) The proof works by case distinction on the reasons for (normal or abrupt) completion of the focused AS. This proof technique is very close to the principle of AE, which consists in reasoning about programs not based on their structure, but on their *effects*. For a complex language like Java, it would be barely feasible to prove the theorem by structural induction on the syntax of the programming language.

(2) We defined the semantics of APEs (see Sect. 4.2.2) as a conjunction of JavaDL formulas. For a fixed, but arbitrary instantiation of the AS in the conclusion, we can assume the validity of this conjunction. The fact that this shares common elements with the rule's premise allows for strong, validity-preserving simplifications.

(3) Using abstract updates and first-order symbols such as exc^{P} *dependently* fresh for an AS is soundness-critical; however, it is admissible since this only happens in the AE rules, and for ASs that are (behaviorally) isomorphic. To renounce dependently fresh first-order symbols would be sound, but either incomplete, or require non-trivial postconditions in the presence of multiple APEs with the same identifier symbol to be complete. The usage of (non-dependent) fresh abstract updates would even

require to specify the whole framed post-state for completeness. Note that all terms with dependently fresh symbols depend on the current value of the relevant part of the context state (the footprint of the AS). The contrary would be unsound.

Details are in Appendix B. □

Theorem 4.2. *The AE rule* abstractExpression *(Fig. 4.2) is sound.*

Proof Sketch. The soundness proof for abstractExpression works analogously to the proof of Thm. 4.1, with the additional observation that the variable v written if, and *only* if, the expression completes normally. □

Theorem 4.3. *The AE rule* abstractStatement *(Fig. 4.1) is complete.*

Proof Sketch. We prove the validity of the premise of rule abstractStatement based on the assumption of the validity of the conclusion. The essence of the proof is that we achieve completeness due to re-using "dependently fresh" logic symbols introduced for ASs with the same identifier symbols. This argument is non-standard: There is no formal connection between interpretations of abstract update symbols and the ASs they have been introduced for, but since there is only one rule for executing ASs and this rule always uses the *same* symbols for ASs with the same identifier symbol, we narrow down the interpretations of introduced logic symbols to the feasible ones.

Details are in Appendix B. □

Theorem 4.4. *The AE rule* abstractExpression *(Fig. 4.2) is complete.*

Proof Sketch. Analogous to Thm. 4.3; for normal completion, it is important that also the value to which the AExp evaluates is chosen dependently fresh, as AExps with the same identifier symbols have to evaluate equally in equal pre-states. □

In the definitions of abstractStatement and abstractExpression, new path condition elements are not directly added to the context Γ, but rather put as the premise of an implication below the scopes of the leading updates. Thus, we save redundancy in the notation, as each update occurs exactly once. This optimization is not possible when describing the rules as SE rules for Symbolic Execution States.

To emphasize that AE is not only implementable in JavaDL, but in any logic-based SE system with an explicit notion of state changes—which can be extended by abstract state changes—we show the rule for AExps in Fig. 4.3.

Note the redundant appearance of updates. We simplified the notation a bit by using update concatenations "$\mathcal{U}_1 \circ \mathcal{U}_2$" instead of "$\mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2$".

$$\frac{\begin{array}{l}\big(C \cup \big\{\{\mathcal{U} \circ (\mathtt{throwsExc} := \mathit{throwsExc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})))\} \\ \qquad (\mathtt{throwsExc} \doteq \mathrm{TRUE} \leftrightarrow \mathit{pre}(\mathit{excSpec})), \\ \qquad \{\mathcal{U} \circ (\mathtt{throwsExc} := \mathit{throwsExc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))) \circ \\ \qquad\quad \mathcal{U}_{\mathsf{e}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint})) \circ \\ \qquad\qquad (\mathtt{exc} := \mathit{exc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})) \,||\, \mathtt{res} := \mathit{res}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})))\} \\ \qquad\qquad\quad ((\mathtt{throwsExc} \doteq \mathrm{TRUE} \rightarrow \mathtt{exc} \not\doteq \mathtt{null} \wedge \mathit{post}(\mathit{excSpec})) \wedge \\ \qquad\qquad\quad\; (\mathtt{throwsExc} \not\doteq \mathrm{TRUE} \rightarrow \mathit{normalPost}))\big\}, \\ \;\mathcal{U} \circ (\mathtt{throwsExc} := \mathit{throwsExc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))) \circ \\ \quad \mathcal{U}_{\mathsf{e}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint})) \circ \\ \qquad (\mathtt{exc} := \mathit{exc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})) \,||\, \mathtt{res} := \mathit{res}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))), \\ \;\pi\ \mathtt{\textbf{if}\ (throwsExc)\ \textbf{throw}\ exc;} \\ \quad \mathtt{v\ =\ res;}\ \omega\big)\end{array}}{(C, \mathcal{U}, \pi\ \mathtt{v{=}\textbf{\textbackslash abstract\_expression}}\ T\ \mathtt{e;}\ \omega)}\ \text{abstractExpressionSE}$$

Figure 4.3: Symbolic Execution Rule for AE of Abstract Expressions

Loops and Method Calls When using loop invariants to abstract from loops with symbolic guards, the induction step and use cases have to be proven in SE states with an *arbitrary* number of previous iterations. This means that part of the context may have been invalidated, and therefore has to be suitably *masked* (or *anonymized*). One has two options: Either, the whole context is erased; consequently, all needed information has to be encoded in the invariant. Alternatively, an *anonymizing update* is added to selectively erase at least the *actually changed* part of the context (the *loop frame*). The latter solution is adopted in KeY and JavaDL. Normally, anonymizing updates have the shape $\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{loopFrame}, \mathit{anonHeap}) \,||\, \mathtt{x}_1 := c_1 \,||\, \ldots \,||\, \mathtt{x}_n := c_n$, where the $\mathtt{x}_i$ are program variables written in the loop, *loopFrame* is the modifier set specified by the user, and *anonHeap* and the c_i as well as *anonHeap* are fresh constants of suitable types.

If loop bodies contain APEs, this is not sufficient. The modifier set *loopFrame* only concerns heap locations, but APEs with abstract dynamic frames may also change arbitrary *program variables*. Therefore, we append an additional *abstract* update to the anonymizing update shown before: $\mathcal{U}_{\mathtt{loop}}(\mathit{abstrLoopFrame} :\approx\ )$ masks all locations in the abstract loop frame *abstrLoopFrame*, which consists of all dynamic frame specification variables existing in the body. Since the masking should be based on "fresh" values, the right-hand side of the abstract update is left empty. The abstract update symbol $\mathcal{U}_{\mathtt{loop}}(\mathit{abstrLoopFrame})$ is created fresh. In the case of *method contracts*, the contract's modifier set has to comprise frame specification variables of APEs in the method's body, because they might, in turn, might represent heap locations. Such specification variables can be extracted and added to the modifier set automatically. We do *not* need abstract updates in the method contract rule, since changes to local variables inside a method are not visible to the outside. Therefore, we can rely on existing mechanisms in KeY.

4.3.2 Rules for Abstract Update Simplification and LocSet Extensions

The addition of abstract updates preserves soundness of almost all simplification rules for *concrete* updates (see Sect. 2.2.3). Those are also used for abstract updates, e.g., to convert sequential update applications to parallel ones. One rule has to be changed, though: We had to strengthen the side condition of rule $\mathsf{dropUpdate}_2$, which drops an elementary update from a parallel one if the assigned program variable is not free in the target. For example, in the formula $\{\mathtt{x} := t_1 \,\|\, \mathtt{y} := t_2\}\mathtt{y} > \mathtt{z}$, we can drop the elementary update $\mathtt{x} := t_1$ (replace it by *Skip*) as $\mathtt{x}$ does not occur in $\mathtt{y} > \mathtt{z}$, or formally, $\mathtt{x} \notin \mathit{fpv}(\mathtt{y} > \mathtt{z})$. However, this is unsound when adding **`\value`** terms depending on dynamic frame specification variables. Consider, for instance, the formula $\mathit{value}(\mathit{locs}) \doteq \{\mathtt{x} := 17\}\mathit{value}(\mathit{locs})$, where *locs* is a constant of type *LocSet*. If we do not know for sure that $\mathtt{x}$ is not in the location set represented by *locs*, the elementary update must not be dropped, and the formula is *not* valid. Consequently, in addition to confirming that $\mathtt{x} \notin \mathit{fpv}(t)$, we also need to check that $\mathit{disjoint}(\mathring{\mathtt{x}}, \mathit{locset})$ holds for each $\mathit{value}(\mathit{locset})$ occurring in t.

The replacement $\mathsf{dropUpdate}_2'$ for $\mathsf{dropUpdate}_2$, as well as our additional simplification rules for abstract updates, therefore not only depend on a *local* condition defined on the target term, but additionally on the *sequent-global context Ctx*, as they have to look for appropriate assumptions justifying the simplification steps. Instead of starting a side proof of "$\mathit{Ctx} \vdash \mathit{condition}$" for verifying these conditions every time an abstract update expression should be simplified, we search the context for literal occurrences of a *normal form* of the conditions. This strongly depends on the existence of such normal forms in the context and requires also considering permutations (e.g., looking for $\mathit{intersect}(s_1,s_2) \doteq \mathit{empty}$ *and* $\mathit{intersect}(s_2,s_1) \doteq \mathit{empty}$), but is much more efficient. If we should fail to discover a condition literally in the context which in fact could be deduced from it, this leads to a simplification not being applicable and is therefore problematic for completeness, but *not soundness-critical*. For creating normal forms, there are JavaDL calculus rules like "disjointDefinition" that, for instance, transform a formula $\mathit{disjoint}(s_1,s_2)$ to the equivalent normal form $\mathit{intersect}(s_1,s_2) \doteq \mathit{empty}$.

We formalize *irrelevance checking* by a predicate $\mathit{irrelevant}(\mathit{Ctx},\mathit{locset},t)$ expressing that the location set *locset* is not relevant for the target t. It holds if the context $\mathit{Ctx} \subseteq \mathrm{Fml}$ contains an assumption $\mathit{intersect}(\mathit{locset},s) \doteq \mathit{empty}$ (or $\mathit{intersect}(s,\mathit{locset}) \doteq \mathit{empty}$) for each *LocSet* constant s such that $\mathit{value}(s)$ is a subterm of t. There are some special cases: The contraction rule for abstract updates introduces placeholders "_" to frames which are by definition "irrelevant", and treated accordingly. For program variables, we assert that the variable does not occur freely in the target. For heap locations, we assert that there is no free occurrence of the heap variable. The latter is a safe overapproximation; fine-grained heap-related simplifications require dedicated rules (we explain an example further below). We define *irrelevant* on *tuples* of locations.

Definition 4.12 (Location Set Irrelevance Checking). Let $\mathit{Ctx} \subseteq \mathrm{Fml}$, $\mathit{locs} \in (\mathrm{Trm}_{\mathit{LocSet}})^n$ and $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$. The predicate $\mathit{irrelevant}(\mathit{Ctx},\mathit{locs},t)$ is defined as

$$\begin{aligned}&\mathit{irrelevant} := \{(\mathit{Ctx},(s_1,s_2,\dots,s_n),t) \mid \forall i=1,\dots,n,\ (s_i = \_ \lor\\ &\quad(\forall \text{ expr. } \mathit{value}(s) \text{ in } t, \mathit{intersect}(s_i,s) \doteq \mathit{empty} \in \mathit{Ctx} \text{ or } \mathit{intersect}(s,s_i) \doteq \mathit{empty} \in \mathit{Ctx}) \land\\ &\qquad\qquad (s_i = \mathring{\mathsf{x}} \rightarrow \mathsf{x} \notin \mathit{fpv}(t)) \land (s_i = (o,f) \rightarrow \mathsf{heap} \notin \mathit{fpv}(t)))\}. \quad \Diamond\end{aligned}$$

In addition to *irrelevant*, which tells us that assigning a location set has *no* effect on the valuation of a target, we need a complementary predicate $\mathit{overwrites}(\mathit{Ctx},\mathit{locs}_1,\mathit{locs}_2)$ expressing that an assignment *of all* ***\hasTo*** *locations* in the location list locs_1 will also assign all locations in the location list locs_2. We can only consider **\hasTo** locations, since locations not designated by that modifier might not be overwritten. Depending on the shape of a location in locs_2, there are several alternatives that allow concluding that the location is overwritten. In the easiest case, the location literally occurs in locs_1, as in $\mathit{overwrites}(\mathit{Ctx},\mathring{\mathsf{x}}^!,\mathring{\mathsf{x}})$ or $\mathit{overwrites}(\mathit{Ctx},\mathit{abstrLocSet}^!,\mathit{abstrLocSet})$; the judgment is independent from *Ctx* in these cases. If the location is a singleton (either a heap location or program variable location), we check whether the corresponding element-of expression, e.g., $\varepsilon(o,f,s)$, is in the context. Otherwise, we have to find some combination of **\hasTo** locations in *Ctx* such that the union of these locations covers the $locs_2$ location.

Definition 4.13 (Location Set Overwriting Checking). Let $Ctx \subseteq \mathrm{Fml}$, $locs_1 \in (\mathrm{Trm}_{LocSet})^n$ and $locs_2 \in (\mathrm{Trm}_{LocSet})^m$, $n, m \in \mathbb{N}$. The predicate $overwrites(Ctx, locs_1, locs_2)$ is defined as

$$\begin{aligned}
overwrites := \big\{&(Ctx, (s^1_1, s^1_2, \ldots, s^1_n), (s^2_1, s^2_2, \ldots, s^2_m)) \mid \forall i = 1, \ldots, m, \big(s^2_i = _ \text{ or} \\
&\exists s^1_k = s^!, \big(\big(s^2_i = s\big) \lor \big(s^2_i = (o,f) \land \varepsilon(o,f,s) \in Ctx\big) \lor \big(s^2_i = \mathring{\mathsf{x}} \land \varepsilon(\mathring{\mathsf{x}}, s) \in Ctx\big)\big) \lor \\
&\exists s^1_{k_1} = (s'_1)^!, \ldots, s^1_{k_l} = (s'_l)^!,\ \big(subset(s^2_i, union(s'_1, union(\ldots, s'_l))) \in Ctx\big)\big)\big\}. \quad \Diamond
\end{aligned}$$

For the definition of our abstract update simplification rules, we assume that they are applied to a subexpression t of a formula φ, written $\varphi(t)$, in one of the situations $\Gamma \vdash \varphi(t), \Delta$ or $\Gamma, \varphi(t) \vdash \Delta$. We then set $Ctx := \bigwedge \Gamma \land \bigwedge_{\psi \in \Delta}(\neg\psi)$.

Figures 4.4 and 4.5 list all simplification rules for abstract updates. We use notational simplifications for lists (which already appeared earlier): For instance, for an n-tuple *frame*, we mean $value(0), \ldots, value(n)$ when writing $value(frame)$ (similarly for the application of updates, etc.). Rule $\mathsf{dropUpdate}'_2$ has already been discussed. There are four more rules $\mathsf{dropUpdate}_3$ to $\mathsf{dropUpdate}_6$ for dropping updates. Rules $\mathsf{dropUpdate}_3$ and $\mathsf{dropUpdate}_4$ correspond to the already existing $\mathsf{dropUpdate}_1$ dropping an earlier update within a parallel composition if it is dominated by a later one. The first of these rules replaces an earlier concrete update $\mathsf{a} := t'$ by *Skip* if a is overwritten by the frame of a later abstract update. The second rule treats the case of an earlier *abstract* update that is dropped. This case is more complex due to the more complex nature of abstract updates. We can drop the abstract update $\mathcal{U}_\mathsf{P}(frame :\approx footprint)$ from a parallel update if there is a series of updates occurring later in the parallel construction which, together, overwrite *frame*. A simple case would be to replace $\mathcal{U}_\mathsf{P}(\mathsf{x} :\approx footprint)$ by *Skip* in $\mathcal{U}_\mathsf{P}(\mathsf{x} :\approx footprint) \,||\, \mathsf{x} := t'$, but for more complicated *frame* expressions, it is not required that a *single* update overwrites all contained locations at once. The rule $\mathsf{dropUpdate}_5$ corresponds to $\mathsf{dropUpdate}_2$, treating the case of an (abstract) update that is dropped since the locations assigned by it are irrelevant for the target term. Two rules address special cases with heap-related expressions. Rule $\mathsf{dropUpdate}_6$ matches an abstract update inside a *select* expression $select_A(\{\ldots \,||\, \mathcal{U}_\mathsf{P}(frame :\approx footprint) \,||\, \ldots\}\mathsf{heap}, o, f)$. If the selected location is not in the frame of the abstract update, i.e., $\neg\varepsilon(o, f, fr_i)$ is in the context for all parts fr_i of *frame*, we can remove the abstract update. In rule $\mathsf{dropAnonInUpdate}$, there is no abstract update. It simplifies heap

$$\{\ldots \,||\, \mathsf{a} := t' \,||\, \ldots\} t \rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \ldots\} t \qquad \mathsf{dropUpdate}_2'$$
where $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$, $\mathit{irrelevant}(\mathit{Ctx}, \mathring{\mathsf{a}}, t)$

$$\{\ldots \,||\, \mathsf{a} := t' \,||\, \ldots \,||\, \mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\} t \qquad \mathsf{dropUpdate}_3$$
$$\rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \ldots \,||\, \mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\} t$$
where $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$, $\mathit{overwrites}(\mathit{Ctx}, \mathit{frame}, \mathring{\mathsf{a}})$

$$\{\ldots \,||\, \mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathcal{U}_1 \,||\, \ldots \,||\, \mathcal{U}_n\} t \qquad \mathsf{dropUpdate}_4$$
$$\rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \mathcal{U}_1 \,||\, \ldots \,||\, \mathcal{U}_n\} t$$
where $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$,
the left-hand side of update $\mathcal{U}_i$ has elements $\mathit{fr}^i_1, \ldots, \mathit{fr}^i_{k_i}$,
$\mathit{overwrites}(\mathit{Ctx}, (\mathit{fr}^1_1, \mathit{fr}^1_2, \ldots, \mathit{fr}^1_{k_1}, \ldots, \mathit{fr}^n_{k_n}), \mathit{frame})$

$$\{\ldots \,||\, \mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathcal{U}\} t \rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \mathcal{U}\} t \qquad \mathsf{dropUpdate}_5$$
where $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$, $\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}, t)$
and $\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}, \mathcal{U})$

$$\mathit{select}_A(\{\ldots \,||\, \mathcal{U}_{\mathsf{P}}(\mathit{fr}_1, \ldots, \mathit{fr}_n :\approx \mathit{footprint}) \,||\, \ldots\}\mathsf{heap}, o, f) \qquad \mathsf{dropUpdate}_6$$
$$\rightsquigarrow \mathit{select}_A(\{\ldots \,||\, \mathit{Skip} \,||\, \ldots\}\mathsf{heap}, o, f)$$
where $\forall i = 1, \ldots, n,\ \neg\varepsilon(o, f, \mathit{fr}_i) \in \mathit{Ctx}$

$$\{\ldots \,||\, \mathsf{heap} := \mathit{anon}(h, \mathit{frame}, \mathit{anonHeap}) \,||\, \mathcal{U}\} t \qquad \mathsf{dropAnonInUpdate}$$
$$\rightsquigarrow \{\ldots \,||\, \mathsf{heap} := h \,||\, \mathcal{U}\} t$$
where $\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}, t)$ and $\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}, \mathcal{U})$

$$\{\ldots \,||\, \mathcal{U}_{\mathsf{P}}(\ldots, \mathit{frame}_i, \ldots :\approx \mathit{footprint}) \,||\, \mathcal{U}\} t \qquad \mathsf{contraction}$$
$$\rightsquigarrow \{\ldots \,||\, \mathcal{U}_{\mathsf{P}}(\ldots, \_, \ldots :\approx \mathit{footprint}) \,||\, \mathcal{U}\} t$$
where $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$, $\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}_i, t)$
and $\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}_i, \mathcal{U})$

Figure 4.4: Abstract Update Simplification Rules

$$\{\ldots \,||\, \mathscr{U}_{\mathsf{Q}}(\mathit{frame}' :\approx \mathit{footprint}') \,||\, \mathscr{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\}t \qquad \mathsf{reorderUpd}_1$$
$$\rightsquigarrow \{\ldots \,||\, \mathscr{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathscr{U}_{\mathsf{Q}}(\mathit{frame}' :\approx \mathit{footprint}') \,||\, \ldots\}t$$
where $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$,
identifier "$\mathscr{U}_{\mathsf{P}}$" is lexicographically smaller than "$\mathscr{U}_{\mathsf{Q}}$",
$\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}', \mathit{value}(\mathit{frame}))$

$$\{\ldots \,||\, \mathsf{a} := t' \,||\, \mathscr{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\}t \qquad \mathsf{reorderUpd}_2$$
$$\rightsquigarrow \{\ldots \,||\, \mathscr{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathsf{a} := t' \,||\, \ldots\}t$$
where $t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}$, $\mathit{irrelevant}(\mathit{Ctx}, \mathring{\mathsf{a}}, \mathit{value}(\mathit{frame}))$

$$\{\mathscr{U}\}\mathscr{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{footprint}) \rightsquigarrow \mathscr{U}_{\mathsf{P}}(\mathit{frame} :\approx (\{\mathscr{U}\}\mathit{footprint})) \qquad \mathsf{applyOnRigid}_9$$

$$\mathscr{U}_{\mathsf{P}}(\ldots, \mathring{\mathsf{x}}^!, \ldots :\approx \mathit{footprint}) \qquad \mathsf{extractHasTo}$$
$$\rightsquigarrow \mathscr{U}_{\mathsf{P}}(\ldots, \_, \ldots :\approx \mathit{footprint}) \,||\, \mathsf{x} := f_k^{\mathsf{P}}(\mathit{footprint})$$
where $\mathring{\mathsf{x}}^!$ occurs at position k within the abstract update,
f_k^{P} is created *dependently fresh* for position k and the identifier $\mathscr{U}_{\mathsf{P}}$

Figure 4.5: Abstract Update Simplification Rules (Continued)

anonymizations that have no effect, since the masked part of the heap is not accessed in subsequent expressions. This occurs frequently in abstract contexts. Since the *irrelevant* predicate does not apply if the `heap` variable occurs freely in the target expression, e.g., if there is a modality in the target, "conventional" non-abstract contexts are not affected.

If only *some* of the left-hand sides of an abstract update are ineffective and the rules $\mathsf{dropUpdate}_4$ and $\mathsf{dropUpdate}_5$ are not available, we have to perform a more fine-grained simplification step than dropping the whole update. The formula

$$\{\mathscr{U}_{\mathsf{P}}(\mathring{\mathsf{x}}, \mathring{\mathsf{z}} :\approx \mathit{footprint})\}\mathsf{z} \doteq \{\mathscr{U}_{\mathsf{P}}(\mathring{\mathsf{y}}, \mathring{\mathsf{z}} :\approx \mathit{footprint})\}\mathsf{z}$$

is semantically valid, but not provable with the rules discussed so far. In this situation, the rule contraction is applicable. It replaces ineffective parts of an abstract update's left-hand side with the "irrelevant" location "_". For our example, this results in

$$\{\mathscr{U}_{\mathsf{P}}(\_, \mathring{\mathsf{z}} :\approx \mathit{footprint})\}\mathsf{z} \doteq \{\mathscr{U}_{\mathsf{P}}(\_, \mathring{\mathsf{z}} :\approx \mathit{footprint})\}\mathsf{z}$$

which is trivially provable. Compared to contraction rules as known from, e.g, tableaux or sequent calculi, we do not *actually* "contract", i.e., remove elements from the left-hand side of the update. Indeed, this would be unsound (cf. Remark 4.3). The symbol "_" receives special treatment in the definitions of the relations *irrelevant* and *overwrites* as it is always considered to be overwritten / not relevant. An abstract update with *only* "_" left-hand sides can be dropped by rules $\mathsf{dropUpdate}_4$ and $\mathsf{dropUpdate}_5$ independently of the context.

JavaDL transforms concrete programs to a sequence of updates which are ultimately applied onto the postcondition. Abstract updates can usually not be applied in this sense. Their left-hand sides, if they are not designated as **`\hasTo`** locations, only constitute "upper bounds" on the locations that will be changed by the update. Moreover, it makes no sense to "apply" an assignment to an *abstract* location set. Generally, in proofs with abstract updates and *value*(*locset*) terms for abstract location sets *locset*, some update applications cannot be simplified away and remain in the leaves. Therefore, especially for the use case of relational program verification, we need to establish a *normal form* within parallel updates. A straightforward example for a situation where this is needed is an equivalence proof for a program and a transformed version after swapping two statements. This normal form is established by the rules $\mathsf{reorderUpd}_1$ and $\mathsf{reorderUpd}_2$. They push abstract updates within parallel updates as much to the front as possible without changing the semantics of the parallel update; however, an abstract update may only be pushed past another abstract update if it has a lexicographically smaller identifier symbol. If there are no conflicts between the elementary abstract and concrete updates within a parallel update, it is normalized to a block of abstract updates ordered according to the lexicographic order of their identifiers, followed by a block of concrete elementary updates.

Although it is generally impossible to apply abstract updates by performing a substitution in the target (which is the ultimate effect of the application of concrete updates), we can (1) apply updates *on* an abstract update, and (2) perform an effective simplification for the special case of program variable locations in the left-hand side marked as **\hasTo**. The corresponding rules, applyOnRigid$_9$ and extractHasTo, are shown in Fig. 4.5. The rule applyOnRigid$_9$, as indicated by the name, belongs to the series of simplification rules pushing update applications further down into the term structure. It specifies that the application of an update $\mathcal{U}$ to an abstract update is equal to the abstract update with the same left-hand side, but $\mathcal{U}$ applied to the footprint. For Item (2), consider the formula

$$\{\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{x}}^! :\approx footprint)\}\varphi.$$

Since the update has to change the value of x (based on value of the term *footprint*), the formula is equivalent to $\{\mathsf{x} := f(footprint)\}\varphi$ for a suitably chosen function symbol f. "Suitably chosen", in this case, means that the function has to be chosen *dependently fresh* for the identifier symbol $\mathcal{U}_{\mathsf{P}}$ of the abstract update to satisfy condition (2) of the semantics of abstract updates (Def. 4.7). This is generalized to abstract updates with multiple left-hand sides by using function symbols f_k^{P} indexed not only with the identifier, but also with the index k of the respective left-hand side. It is not always feasible to convert a whole abstract update to a concrete one, as in the following variation of the example:

$$\{\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{z}}, \mathring{\mathsf{x}}^!, locset :\approx footprint)\}\varphi$$

The assignable program variable location "$\mathring{\mathsf{z}}$" is not marked as **\hasTo**, and the abstract location set *locset* cannot be converted to a concrete update. Therefore, we extract **\hasTo** program variable locations individually and replace their positions in the left-hand side of the abstract update by the irrelevant location "_". Note that we could also only remove the $\circ^!$ specifier and leave it to the contraction rule to replace the assignable by "_", which would require one more simplification step. Our simplification rule extractHasTo incorporates these considerations. Applying extractHasTo to the example above yields

$$\{\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{z}}, \_, locset :\approx footprint) \,||\, \mathsf{x} := f_2^{\mathsf{P}}(footprint)\}\varphi.$$

Rule for LocSet Extensions To complete our deductive framework for AE, we present the rules for the extensions of the *LocSet* theory introduced in Sect. 4.2.1. These simple rules are displayed in Fig. 4.6. The first one, dropHasTo, formalizes the semantics of the $\circ^!$ specifier, which equals the meaning of its argument. It cannot be applied to remove has-to specifiers in abstract update left-hand sides, since those are *parts of an operator* and not individual terms. Four rules named "valueOf..." address *value* terms. For program variable locations, those resolve to the corresponding program variable; heap locations

$$locset^{!} \rightsquigarrow locset \qquad \text{dropHasTo}$$

$$value(\mathring{\mathsf{x}}) \rightsquigarrow \mathsf{x} \qquad \text{valueOfSingletonPV}$$

$$value(o, f) \rightsquigarrow select_A(\mathtt{heap}, o, f) \qquad \text{valueOfSingleton}$$

$$value(anonPV(\mathring{\mathsf{x}}, locset, \mathring{\mathsf{y}})) \rightsquigarrow \mathsf{y} \qquad \text{valueOfAnonPV}_1$$
$$\text{where } \varepsilon(\mathring{\mathsf{x}}, locset) \in Ctx$$

$$value(anonPV(\mathring{\mathsf{x}}, locset, \mathring{\mathsf{y}})) \rightsquigarrow \mathsf{x} \qquad \text{valueOfAnonPV}_2$$
$$\text{where } \neg\varepsilon(\mathring{\mathsf{x}}, locset) \in Ctx$$

$$heapLocs(o, f) \rightsquigarrow (o, f) \qquad \text{heapLocs}_1$$

$$heapLocs(\mathring{\mathsf{x}}) \rightsquigarrow empty \qquad \text{heapLocs}_2$$

$$heapLocs(empty) \rightsquigarrow empty \qquad \text{heapLocs}_3$$

$$heapLocs(\circ(s_1, s_2)) \rightsquigarrow \circ(heapLocs(s_1), heapLocs(s_2)) \qquad \text{heapLocs}_4$$
$$\text{where } \circ \in \{union, intersect, setMinus\}$$

$$pvLocs(\mathring{\mathsf{x}}) \rightsquigarrow \mathring{\mathsf{x}} \qquad \text{pvLocs}_1$$

$$pvLocs(o, f) \rightsquigarrow empty \qquad \text{pvLocs}_2$$

$$pvLocs(empty) \rightsquigarrow empty \qquad \text{pvLocs}_3$$

$$pvLocs(\circ(s_1, s_2)) \rightsquigarrow \circ(pvLocs(s_1), pvLocs(s_2)) \qquad \text{pvLocs}_4$$
$$\text{where } \circ \in \{union, intersect, setMinus\}$$

Figure 4.6: Rules for LocSet Extensions

are transformed to $select_A$ terms, where A is the type of the referred field. Apart from that, there are two rules for a *value* application to terms created with the *anonPV* operator. There are no rules for terms *value*(*locset*) with abstract location sets *locset*; update applications accumulate in front of these expressions and are not removed if *locset* might be relevant for their left-hand sides. The remaining eight rules capture the semantics of the *heapLocs* and *pvLocs* filter functions returning heap or program variable locations of a *LocSet* term. They are defined recursively for compound terms built by *union*, *intersect* and *setMinus*.

4.4 Implementation

Abstract Execution has been implemented for the heavyweight SE engine of the deductive program prover KeY [Ahr+16]. Table 4.5 displays some code metrics about the implementation, which comprises 3,235 lines of Java code and 397 lines of KeY taclets. In contrast to the descriptions in Sect. 4.3, there is more than one rule for ASs: We implemented different rules for different *contexts* (cf. Remark 4.9). For instance, if an AS is executed outside of any loop context, **`continue`**s and unlabeled **`break`**s do not make sense, which is why they are not triggered by our context-sensitive rules. Alternatively, specifications had to explicitly exclude abrupt completion modes whenever they do not make sense.

AE Rules as Taclets We decided to implement our abstract execution rules not as built-in Java rules, as is customary for complex rules in the KeY system, but as *taclets* in KeY's taclet syntax (cf. Sect. 2.2.7). There are four rules for ASs and one for AExps (since the only reason why an AExp might complete abruptly is due to a thrown exception, which is meaningful in any context). Taclets have several advantages; most importantly, they are, compared to Java classes, easier to read and maintain, due to the restricted syntax. Thus, it was easy to support additional completion modes or add additional premises. Future changes to the rules are likewise made easier. Should we, for example, decide to only have a single rule for ASs, this could be accomplished by removing all rules but the most complex one and removing from that rule the conditions restricting its applicability to the respective proof context. Our AE taclets are the most complex ones that, to the best of our knowledge, have ever been implemented. The longest rule for ASs has 19 variable conditions, a "**`\replacewith`**" clause of 68 lines and spans 79 lines of code in total. To be able to write those taclets as they are, we had to significantly extend the existing taclet language. This involves extensions to the parser definition files (which are not included in the numbers of Table 4.5) as well as the 11 Java classes for additional variable conditions and transformers amounting to 598 lines of code in the table. The rationale

Table 4.5: Code Metrics for the AE Implementation

Java Code				
Description	Files	Lines of Code	Lines of Comments	Total
Abstract Updates	14	988	713	1,701
Abstract Update Simplification Rules	8	929	488	1,417
Variable Conditions and Transformers for Taclets	11	598	265	863
Utility Classes	3	445	532	977
APEs	4	275	405	680
SUM	40	3,235	2,403	5,638

Taclets				
Description	Taclets	Lines of Code	Lines of Comments	Total
Abstract Execution Rules	5	325	20	345
Abstract Update Rules	2	49	5	53
Locset Rules	4	23	2	25
SUM	11	397	27	423

behind these extensions and the rather high number of variable conditions / transformers is that we wanted to avoid delegating too many responsibilities to few but powerful variable conditions / transformers; instead, our goal was to implement *small pieces* of additional Java code with *clear responsibilities*, exposing as many details as possible in the textual representation of the taclet itself. One of the extensions is a "for-each" construct for iterating over schema variables of list type. The following taclet code occurs in the Java block of the "**\replacewith**" part of the rules for ASs:

```
#foreach (#v1, #label in #vars, #labels) {
    if (#v1) {
        break #label;
    }
}
```

Implementing the for-each construct was more difficult than just implementing a Java class for a transformer like "`#handle-labeled-breaks(#vars, #labels)`" replacing the code above, but the result is more transparent and reusable, and comes closer to the description of the rule in text-book style. In Appendix C, we show two taclets for ASs and AExps as they are actually implemented in KeY.

Built-In Rules for Abstract Update Simplification Unfortunately, we could only implement two of the abstract update simplification rules ($\mathsf{applyOnRigid}_9$ and $\mathsf{extractHasTo}$) as taclets; the "drop" and "reorder" rules are implemented as built-in rules in plain Java code. The main reason for this is that these rules depend on a *variable number of premises* in the context which is *initially unknown* (for instance to implement rule $\mathsf{dropUpdate}_6$). The most elegant approach to realize this would consist in a more flexible syntax for the "**\assumes**" clauses, which seemed to be very difficult to realize. A more feasible alternative is to provide variable conditions with general access to the surrounding sequent. We refactored KeY accordingly; at the end, however, we decided against following this approach, since this would, without further adaptions, impede explicit visualizations in the GUI of the part of the context that was used in simplifications. This is no problem in built-in rules: When applying an abstract update simplification rule depending on the context, all formulas in the premise that have been used to justify the simplification step are highlighted as used assumptions. Nonetheless, the mentioned refactoring would theoretically facilitate an implementation of these rules as taclets. Note that there is not one built-in rule / Java class for each simplification rule listed in Sect. 4.3.2. Instead, there are two classes (one for the rule and one for the corresponding rule application object) for the “drop” and “reorder” rules, each. Thus, we avoid redundancy.

Implementation of Abstract Updates Abstract updates are implemented as KeY “operators”. Each such operator has a fixed list of left-hand sides (the assignable locations of the abstract update) and knows the APE identifier it has been introduced for. For the abstract syntax of the left-hand sides, there exists a dedicated type hierarchy implementing an interface named “`AbstractUpdateLoc`”. Most importantly, abstract updates are only created via a factory class `AbstractUpdateFactory`, which is responsible for creating and re-using the same symbols for the same APEs and left-hand sides, and takes care that such symbols are not re-used “accidentally”. The class is also responsible for creating the “characteristic functions” needed for rule extractHasTo and for converting location set terms to the `AbstractUpdateLoc` hierarchy. Each proof object knows exactly one object of type `AbstractUpdateFactory`.

4.5 Summary and Discussion

We introduced Abstract Execution, a reasoning technique for *behavioral properties of abstract programs*. Behavioral properties are concerned with observable, external effects of a program’s execution. For instance, a program might *change the surrounding state* or *complete abruptly* for a number of reasons. The property that a program is, e.g., free of loops and method calls, is *syntactic* and *not* behavioral. *Abstract programs*, in our framework, contain Abstract Program Elements (Abstract Statements or Abstract Expressions), which serve as placeholders for generally infinitely many concrete programs.

The restriction to external effects sets the stage for an *automatable* framework. In contrast to related approaches (e.g., [ARS05; GS13; KTL09]), ours allows expressing more subtle behavioral properties by providing a versatile *specification language* defining the observable behavior of abstract programs. In particular, one can constrain the locations which an APE may write and read (its *frame* and *footprint*), when and how it completes abruptly, and, for every “completion mode”, what functional postconditions are assured for the resulting state. This allows for powerful specifications; e.g., one can specify that an AS throws an exception exactly if some AExp would return “**`true`**” in the same state.

We use *dynamic frames* (specification variables representing unknown sets of locations) to trade off precision and generality: The frame of an APE can be given a name, admitting all possible instantiations; however, we can impose *constraints on dynamic frames*, typically, e.g., that the frame of one APE is disjoint from the frame or footprint of another APE.

AE is embedded in Java Dynamic Logic, a program logic for Java supporting logic-based Symbolic Execution. Based upon JavaDL's concept of *updates*, syntactic representations of state transitions, we define *abstract updates*. Those represent many state transitions at once and can use dynamic frames as assignment targets. We transform abstract programs to abstract updates and represent abrupt completion by explicit SE branches.

We defined syntax and semantics of APEs, abstract program fragments, and abstract updates. An important contribution is a definition of SE rules for abstract statements, which we *implemented* for the KeY system. The implementation is based on *taclets* as opposed to hard-coded Java rules. This is uncommon for complicated, "high-level" rules, and required extensions of the taclet language. The reward is increased readability, maintainability and extensibility of the AE calculus. In addition, we implemented rules for simplifying abstract updates and for our extensions of KeY's theory of dynamic frames.

Some possible extensions should be investigated in the future; for instance, it might make sense to define separate frames and footprints per completion mode, s.t. an APE assigns, e.g., fewer locations when throwing an exception. Another idea in this regard is to separate frames into a local and heap part. This would add some scoping information, of which AE currently is absolutely oblivious. For example, the local frame could be cleared when an abstract update escapes a method scope.

Especially because we heavily use updates, it is sensible to ask whether AE can be used outside the ecosystem of KeY. We think that a full implementation of AE requires an *explicit notion of state*, in particular, to allow references to dynamic frames; updates proved to be ideal to that end. It was also helpful that KeY already had an implementation of the theory of dynamic frames (the "*LocSet*" theory). Yet, AE is more than our set of calculus rules. It is also a *reasoning principle*: If we are only concerned with the external behavior of an abstract program, we can conduct proofs using a mixture of *abstraction and (small) case distinctions* instead of, e.g., structural induction. This general idea reaches further than KeY; It could also be used to simplify proofs in interactive provers like Coq or Isabelle.

The practical applicability of AE is demonstrated in Chapter 6, where it is used to automatically prove the correctness of refactoring rules (such as "*Slide Statements*" which we discussed in Example 4.4), including some with loops. We need all the power of our specification language to express the sometimes subtle preconditions ensuring safe refactorings. In Sect. 6.4, we also discuss *performance* aspects.

5 Modal and Symbolic Trace Logic

Since the foundational work on program verification during the 1960s [Flo67; Hoa69], the program verification tasks that were studied have much broadened beyond mere functional (partial or total) correctness. Basic variations include termination [Häh+86], reachability [RHS95], and program synthesis [Hei92; Smi90]. Starting in the early 2000s, verification of *relational* properties of programs [BU18], such as information flow [DHS05; SM03], correct compilation [Ler09], or correctness of program transformations (refactoring) [GM06] has been in the focus of interest. Relational properties compare two programs having similar behavior. It is even more challenging to reason about programs having related, but intentionally *differing* behavior, such as in program evolution [GS13].

For all these tasks, *dedicated* verification approaches were developed: dynamic logic [HTK00], Hoare quadruples [Yan07], self composition [BDR04; DHS05], product programs [BCK11], etc. Usually, the verification problem to be solved is stated informally, and then the problem is *directly* formalized in the approach to be used for its solution. Hence, the formalism that a problem is stated in and the formalism where it is solved, are *conflated*. We consider this problematic for two reasons:

(1) **Premature commitment to a specific solution approach.** If one has invested to master a specific methodology, the temptation to solve *any* problem by modifying or extending the familiar is considerable, even if a different approach would have been more efficient, flexible, or easily extensible.

(2) **Hard to detect commonalities and to transfer results.** Detecting structural similarity between different problem areas facilitates transferring insights and solutions from one problem space to another one. In formal verification, this additionally opens the road to re-use of software tools for new tasks. To be able to spot commonalities, it is essential to know which aspects of a problem are genuinely new and hence require a novel approach. However, if a problem is *already formalized* in terms of a specific solution method, it is hard to identify commonality and analogy.

In [SH19b], we proposed a framework based on *three re-occurring principles*:

(1) *abstraction* of program runs in the sense of abstract interpretation [CC77];

(2) *approximation* of a set of program runs by a superset;

(3) the capability to handle *abstract programs*.

Abstraction makes it possible to compare programs written in different languages via a suitable abstraction of their traces. Approximation is needed to focus on a specific property and "forget" irrelevant information.[1] Finally, to reason about program transformation (synthesis, compilation, refactoring, etc.) it is essential to be able to define programs with *unspecified parts*. The tool we use to realize this last principle, and thus a core constituent of the system, is Abstract Execution. Based on these principles, one can express a wide variety of verification problems in a uniform, comparable manner.

We propose *Modal Trace Logic (MTL)*, a uniform logic system which allows to integrate constructs from various *Trace Description Languages (TDLs)*. A TDL is anything with a *trace semantics*, including (abstract and concrete) programs and formulas of different logics, e.g., first-order or temporal logic. In the case of programs p, this means that for an initial execution state σ we can obtain the set of all traces ("program runs") that are possible when p is started in σ. The trace semantics of a first-order formula φ are all traces starting in an initial state with final states satisfying φ, for a Linear Temporal Logic (LTL) formula $\Box\xi$ all traces of states satisfying ξ, and so on.

The "modal" part of our framework is contributed by the *trace modality* $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$, which we designed according to the three principles explained above. It is *valid* if the traces of the *implementation* $\mathscr{C}_{impl}$ are *approximated* by the traces of the *specification* $\mathscr{C}_{spec}$ after the *abstraction* step defined by α. Implementation and specification are constructs of arbitrary TDLs. Usually, there is a program on the left, and a program *or* a formula on the right. MTL satisfies some, but not all properties of modal and dynamic logic; in particular, the axiomatic system for (propositional) dynamic logic is not suitable for MTL.

The *only* fixed syntactic construct of MTL, the trace modality, is parametric in the abstraction and TDLs. Consequently, it is not possible to equip the logic with a calculus without instantiating it first to concrete languages and abstractions. Instead, we propose *Symbolic Trace Logic (STL)*, a separate logic working on *symbolic* traces. MTL and STL are connected via constraints on the *translations* from MTL expressions to STL symbolic traces. Thus, one can derive the validity of an MTL formula from a successful proof of its symbolic translation in STL. STL has a *sequent calculus* for reasoning about inclusion of symbolic traces. The calculus consists of a small set of rules and is sound, but *incomplete*:

[1] One can view approximation as a special case of abstraction. Since approximation can be expressed by a subset relation alone, it is unnatural to conflate them, however.

There are symbolic trace inclusions that cannot be shown in STL. The attractive properties of the calculus include that it can distinguish finite from potentially infinite traces, has a modular soundness proof (one argument for each rule), and is, in our opinion, general enough to successfully prove a number of interesting inclusions. The core of the calculus is a *symbolic state subsumption checker*. We provide two concise definitions for weak and strong subsumption of symbolic states, and prove that the latter implies the former.

This chapter is structured as follows. Syntax and semantics of MTL is described in Sect. 5.1. Sect. 5.2 derives a diamond version of the modality and attempts a characterization by investigating which of the classic properties of modal and dynamic logic are true in our system. In Sect. 5.3, we formalize various verification tasks using the trace modality to demonstrate its expressiveness. Finally, Sect. 5.4 describes STL.

5.1 Modal Trace Logic: Syntax and Semantics

Modal Trace Logic is a system for reasoning about sets of execution traces. Traces can be described in different Trace Description Languages (TDLs), like programs and postconditions. The logic does not impose specific languages; in particular, it is not bound to specific *programming* languages. We only require that TDLs need to have a *trace semantics*, which defines the trace set described by expressions (which we frequently call "constructs") of the language. An MTL formula is valid iff it represents, for *each initial state*, the entirety of traces. The only *fixed syntactic element* of MTL is the *trace modality* $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$, where the *implementation* $\mathscr{C}_{impl}$ and *specification* $\mathscr{C}_{spec}$ are constructs of (the same or different) TDLs and α is a *trace abstraction*. The modality expresses that the specification is an approximation of the implementation relative to α. Apart from that, TDLs can be integrated on-demand. For instance, we define a trace semantics for first-order postconditions and LTL formulas. Furthermore, it is easy to integrate propositional junctors with a set-based semantics: The meaning of $\varphi \oplus \psi$ ("φ or ψ") is the union of the traces for φ and ψ. Constituents φ and ψ do not have to stem from the same TDL; φ could be a program and ψ an LTL formula or another program.

We first define some basic notions about traces, introduce our concept of trace abstraction and provide some example abstractions. Afterward, we delineate the logic itself, and exemplarily show chosen TDL definitions.

5.1.1 Traces and Abstractions

Traces are sequences of *states*. A state maps program variables to values of suitable type (see Sect. 2.2, Def. 2.7). In this chapter, we use, apart from the notion of state, also other concepts of JavaDL to avoid redundant definitions (this does not imply that we are tied to the Java language to any degree). We interpret, e.g., first-order terms t by $val(K, \sigma, \beta | t)$.

Definition 5.1 (Traces)**.** A *trace* τ is a (potentially infinite) sequence of states. We write $s_0 s_1 \cdots s_n$ for finite and $s_0 s_1 \cdots$ for finite or infinite traces. The empty trace is denoted by ε. For the set of all traces, we write $\mathit{Traces} = (\mathcal{S})^* \cup (\mathcal{S})^\omega$. The predicate $\mathit{finite}(\tau)$ holds for finite traces, and $\mathit{first}(\tau)$, $\mathit{last}(\tau)$ select a trace's first and final state, where the latter is only defined for finite traces. When writing $\tau\tau'$, we implicitly assume that τ is finite. $\Diamond$

We call an operator on trace sets a *trace abstraction* if it is *monotone*, i.e., every input trace also is an output trace, *idempotent*, i.e., double applications do not change the result, and *homomorphic* on unions and empty sets. Formally, we define:

Definition 5.2 (Trace Abstraction)**.** A *trace abstraction* is a total function α from 2^{Traces} to 2^{Traces} satisfying (1) *monotonicity:* for all $\mathcal{T} \subseteq \mathit{Traces}$, $\mathcal{T} \subseteq \alpha(\mathcal{T})$, (2) *idempotence:* for all $\mathcal{T} \subseteq \mathit{Traces}$, $\alpha(\alpha(\mathcal{T})) = \alpha(\mathcal{T})$, (3) *homomorphic on unions*: for all $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathit{Traces}$, $\alpha(\mathcal{T}_1 \cup \mathcal{T}_2) = \alpha(\mathcal{T}_1) \cup \alpha(\mathcal{T}_2)$, (4) *homomorphic on empty sets*: $\alpha(\emptyset) = \emptyset$. $\Diamond$

Relation to Abstractions in Static Analysis Our notion of trace abstraction conforms to the abstraction concept known from static program analysis and abstract interpretation [CC77; NNH99]. Observe that $\left(2^{\mathit{Traces}}, \subseteq\right)$ is a complete lattice with supremum Traces and infimum $\emptyset$. We can define a "concretization" γ as the identity function on trace sets. Then, $(2^{\mathit{Traces}}, \alpha, \gamma, \{\alpha(\tau) \mid \tau \subseteq \mathit{Traces}\})$ is a *Galois connection* (cf. [NNH99]): It holds that $\gamma(\alpha(\mathcal{T})) = \alpha(\mathcal{T}) \supseteq \mathcal{T}$ since α is monotone (it trivially holds for $\emptyset$). Also, it is true that $\alpha(\gamma(\mathcal{T})) = \alpha(\mathcal{T}) \subseteq \mathcal{T}$ since there has to be some $\mathcal{T}'$ s.t. $\mathcal{T} = \alpha(\mathcal{T}')$, and $\alpha(\alpha(\mathcal{T}')) = \alpha(\mathcal{T}')$ holds because α is idempotent ($\alpha(\emptyset) \subseteq \emptyset$ is also true since α preserves the empty set). This implies that we may lose precision, but not safety, when repeatedly abstracting and concretizing trace sets, which is the intention for the use of Galois connections. Those considerations are meant to motivate some of the required properties of trace abstractions (monotonicity, idempotence and preservation of empty sets). We will not use the Galois connection defined above in the remainder of this chapter.

We introduce some abstractions that are interesting for the applications in Sect. 5.3.

Identity abstraction: $\alpha_{id}(\mathcal{T}) = \mathcal{T}$. We omit α in the notation if it is α_{id}.

Big-step abstraction: $\alpha_{big}(\mathscr{T}) = \{\tau \in \mathit{Traces} \mid \tau' \in \mathscr{T} \land \mathit{first}(\tau) = \mathit{first}(\tau') \land (\neg \mathit{finite}(\tau) \lor \mathit{last}(\tau) = \mathit{last}(\tau'))\}$, i.e., all traces starting with the same states as the traces in the input set which are either infinite or also end with the same states as the inputs. This definition abstracts away from nonterminating traces; replacing "$\neg\mathit{finite}(\tau)$" by "$\neg\mathit{finite}(\tau')$" allows for reasoning about termination. We write α_{big}^{inf} for the big-step abstraction with abstraction of infinite traces and α_{big}^{fin} for the version without.

Obervation abstraction: Let $obs \subseteq \mathrm{PVSym}$ and $\sigma \in \mathscr{S}$. Then, $\sigma \downarrow obs := \{\sigma' \in \mathscr{S} \mid \forall \mathsf{x} \in obs, \sigma(\mathsf{x}) = \sigma'(\mathsf{x})\}$ is the abstraction from all locations in a state that are not in the set *obs*. We define the *observation abstraction* relative to *obs* as

$$\alpha_{obs}(\mathscr{T}) = \{\sigma_1\sigma_2\cdots \mid \sigma_1'\sigma_2'\cdots \in \mathscr{T} \land \sigma_i \in (\sigma_i' \downarrow obs)\}.$$

For a concrete set of variables, for instance $\{\mathsf{x}\}$, we write $\alpha_{\{\mathsf{x}\}}$.

Data abstraction: Let α be an abstraction operator on data types in the sense of *abstract interpretation* [CC77], and γ the corresponding concretization function. We define the *data abstraction* of a set of traces as

$$\alpha_d(\mathscr{T}) = \{\sigma_1\sigma_2\cdots \mid \sigma_1'\sigma_2'\cdots \in \mathscr{T} \land \sigma_i \in \gamma(\alpha(\sigma_i'))\},$$

where the state $\alpha(\sigma)(\mathsf{x}) = \alpha(\sigma(\mathsf{x}))$ is defined pointwise and the state set $\gamma(S)$ is defined as $\{\sigma \mid \sigma' \in S \land \forall \mathsf{x}, \sigma(\mathsf{x}) \in \gamma(\sigma'(\mathsf{x}))\}$.

Combination: Combine two abstractions α_1, α_2 by function composition $\alpha_1 \circ \alpha_2$.

5.1.2 Trace Valuation, Trace Description Languages, and Trace Modality

The semantics of our framework is based on the *trace valuation function* $\mathit{tval}(K, \sigma, \beta | \mathscr{C})$ projecting, for a structure K, initial state σ and variable assignment β, its argument $\mathscr{C}$ to a set of traces. For expressions without free logic variables, we write $\mathit{tval}(K, \sigma | \mathscr{C})$. *A Trace Description Language is a formal language on which tval is defined.*

We define the trace semantics for common TDLs.

First-order postconditions: The trace semantics of $\varphi \in \mathrm{Fml}$ is defined as

$$\mathit{tval}(K, \sigma, \beta | \varphi) = \{\tau \in \mathit{Traces} \mid \mathit{first}(\tau) = \sigma \land \mathit{val}(K, \mathit{last}(\tau), \beta | \varphi) = \mathit{tt}\},$$

i.e., all traces starting in σ whose final states satisfy φ.

LTL formulas: For LTL formulas ξ, the trace semantics $\mathit{tval}(K, \sigma, \beta | \xi)$ is the standard trace semantics for temporal logic. We consider a first-order fragment with atoms of the form φ^{ltl} for first-order formulas φ, and compound operators $\bigcirc\xi$ ("ξ has to hold in the next state"), $\Diamond\xi$ ("ξ has to hold eventually"), $\Box\xi$ ("ξ always has to hold"), and $\zeta\mathsf{U}\xi$ ("ζ has to hold at least until ξ becomes true, which must eventually be the case). The trace semantics of this fragment is defined as:

$$\begin{aligned}
tval\big(K,\sigma|\varphi^{ltl}\big) &:= \{\sigma\tau \mid \tau \in Traces \wedge val(K,\sigma|\varphi) = tt\} \\
tval(K,\sigma|\bigcirc\xi) &:= \{\sigma\tau \mid \tau \in tval(K,\sigma|\xi)\} \\
tval(K,\sigma|\Diamond\xi) &:= \big\{\sigma\tau\tau' \mid \tau,\tau' \in Traces \wedge \tau' \in tval(K,\sigma|\xi)\big\} \\
tval(K,\sigma|\Box\xi) &:= \{\sigma_1\sigma_2\cdots \mid \sigma_1 = \sigma \wedge \forall i \geq 1;\ \sigma_i\sigma_{i+1}\cdots \in tval(K,\sigma|\xi)\} \cup \{\varepsilon\} \\
tval(K,\sigma|\zeta\mathsf{U}\xi) &:= \{\sigma_1\sigma_2\cdots\sigma_k\tau \in Traces \mid \sigma_1 = \sigma \wedge k \geq 0 \wedge \\
&\qquad (\forall 1 \leq i \leq k;\ \sigma_i \in tval(K,\sigma|\zeta)) \wedge \tau \in tval(K,\sigma|\xi)\}
\end{aligned}$$

Java programs: As in the case of ϱ in Sect. 2.2, we leave the definition of the trace semantics for Java programs underspecified and assume that, for a Java program p, $tval(K,\sigma|p)$ returns the set of traces of p starting in σ.

Abstract programs: The semantics of an abstract program fragment $\mathscr{F}$ is defined as $tval(K,\sigma,\beta|\mathscr{F}) = \bigcup_{p\in[\![\mathscr{F}]\!]} tval(K,\sigma,\beta|p)$ (cf. Sect. 4.2).

MTL is a "plugin logic" integrating arbitrary TDLs. Let, for instance, $\mathscr{C}_1 \odot \mathscr{C}_2$ denote the intersection of the trace sets for TDL constructs $\mathscr{C}_1$ and $\mathscr{C}_2$. Then, we can describe by the MTL formula "$(\texttt{x>0})^{ltl} \odot \texttt{x+=1;}$" the set of two-state traces $\sigma\sigma'$ where `x` is strictly positive in σ and its value in σ' is one bigger than its value in σ. In this example, $(\texttt{x>0})^{ltl}$ is from the TDL of LTL formulas, $\odot$ is a general operator on TDL constructs in the sense of a propositional junctor, and `x+=1;` is a program with the usual trace semantics.

Each instantiation of MTL contains *trace modality formulas* $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$, where the implementation $\mathscr{C}_{impl}$ and specification $\mathscr{C}_{spec}$ again stem from arbitrary TDLs. We define the syntax of trace modality formulas.

Definition 5.3 (Trace Modality). Let $\mathscr{C}_{impl}$ and $\mathscr{C}_{spec}$ be two TDL constructs. The *trace modality* is a formula

$$[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$$

where $\alpha \in 2^{Traces} \to 2^{Traces}$ is a trace abstraction. $\Diamond$

The shape of the trace modality, which does not look like a classic "modality", is a mere syntactic difference. Instead of writing $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$, we could write $[\mathscr{C}_{impl}]_\alpha\mathscr{C}_{spec}$. We decided against this variant to clearly scope the specification part. Moreover, since $\mathscr{C}_{spec}$

can also be a program, we consider it more beautiful to confine it to inside the modality's box, which is common in dynamic logic, too. In the former notation, it is also more explicit that the abstraction α is applied to the left *and* the right-hand side of the modality.

The meaning of a trace modality formula $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ is the set of traces which, if they are traces of $\alpha\big(tval\big(K,\sigma|\mathscr{C}_{impl}\big)\big)$, are also traces of $\alpha\big(tval\big(K,\sigma|\mathscr{C}_{spec}\big)\big)$. This includes those that are *not* in $\alpha\big(tval\big(K,\sigma|\mathscr{C}_{impl}\big)\big)$. In other words, the *specification approximates the implementation relative to the abstraction*.

Definition 5.4 (Semantics of the Trace Modality)**.** Let $\mathscr{C}_{impl}$ and $\mathscr{C}_{spec}$ be TDL constructs, and α be a trace abstraction. The semantics of a trace modality formula is defined as

$$\begin{aligned} & tval\big(K,\sigma,\beta|[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]\big) \\ & \quad = \overline{\alpha\big(tval\big(K,\sigma,\beta|\mathscr{C}_{impl}\big)\big)} \cup \alpha\big(tval\big(K,\sigma,\beta|\mathscr{C}_{spec}\big)\big) \\ & \quad = \big\{\tau \in Traces \mid \text{if } \tau \in \alpha\big(tval\big(K,\sigma,\beta|\mathscr{C}_{impl}\big)\big) \text{ then } \tau \in \alpha\big(tval\big(K,\sigma,\beta|\mathscr{C}_{spec}\big)\big)\big\}. \end{aligned} \qquad \Diamond$$

Note the uniformity of Def. 5.4, which does not differentiate between concrete programs, abstract programs, and formulas. This is possible since all MTL atoms (i.e., TDL constructs) have a semantics based on trace sets. In JavaDL, for instance, this is different: formulas evaluate to a truth value and programs to state transitions.

We define notions of *satisfiability* and *validity* for MTL. To distinguish "conventional" validity (based on valuation function *val*) from trace-based validity (based on trace valuation function *tval*), we write "$|\!\models$" instead of "$\models$" for the satisfiability relation.

Definition 5.5 (Trace-Based Satisfiability and Validity)**.** Let $\mathscr{C}$ be a TDL construct. We write $K,\sigma \mid\!\models \mathscr{C}$ and say that $\mathscr{C}$ is *satisfiable* iff $tval(K,\sigma|\mathscr{C}) = Traces$, i.e., if, given a structure K, every possible trace is in the set of traces for $\mathscr{C}$ starting in σ. If $K,\sigma \mid\!\models \mathscr{C}$ for all K and σ, we write $|\!\models \mathscr{C}$ and say that $\mathscr{C}$ is (universally) *valid*. $\Diamond$

In [SH19b], the semantics of the trace modality is directly defined as a truth value and not embedded in a trace-based framework. There, $K,\sigma \mid\!\models [\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ holds if

$$\alpha\big(lift_l^K(\sigma)(\mathscr{C}_{impl})\big) \subseteq \alpha\big(lift_r^K(\sigma)(\mathscr{C}_{spec})\big)$$

where $lift_{l/r}^K(\sigma)$, called "lifting functions" for the *l*eft (implementation) and *r*ight (specification) constructs, generate trace sets for $\mathscr{C}_{impl}$ and $\mathscr{C}_{spec}$ starting in the initial state σ.

After re-writing this to use the trace valuation function *tval*, it amounts to

$$\alpha\left(tval\left(K,\sigma|\mathscr{C}_{impl}\right)\right) \subseteq \alpha\left(tval\left(K,\sigma|\mathscr{C}_{spec}\right)\right)$$

The advantage of the semantics in [SH19b] is that the *approximating* nature of the trace modality is made more explicit by the use of set inclusion: The traces of the specification are a superset of, i.e., approximate, the traces of the implementation. Def. 5.4, in turn, is a uniform extension of the semantics of the box modality of Propositional Dynamic Logic (PDL) [HTK00] to the trace semantics setting (augmented by the use of abstraction and initial states). In PDL, the semantics of a box modality formula is the set of *states* satisfying the property that *if* they are initial states of a terminating transition of the program in the box, *then* the final states are elements of the semantics of the postcondition. The semantics of a PDL program are the pairs of initial and final states. In our framework, *all* TDL constructs are semantically represented by *trace sets*.

Validity of the trace modality according to Defs. 5.4 and 5.5 and Reference [SH19b] is equivalent (and we can therefore use them interchangeably):

Lemma 5.1 (Equivalence of Trace Modality Satisfiability Notions)**.** *Let* $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ *be a trace modality formula, K a structure and $\sigma \in \mathscr{S}$. The following equivalence holds:*

$$\begin{aligned} &\overline{\alpha\left(tval\left(K,\sigma,\beta|\mathscr{C}_{impl}\right)\right)} \cup \alpha\left(tval\left(K,\sigma,\beta|\mathscr{C}_{spec}\right)\right) = Traces \\ \Longleftrightarrow\ &\alpha\left(tval\left(K,\sigma|\mathscr{C}_{impl}\right)\right) \subseteq \alpha\left(tval\left(K,\sigma|\mathscr{C}_{spec}\right)\right). \end{aligned}$$

Proof. Follows from basic set theory and the fact that *Traces* is the universe of traces. □

In the following section, we introduce a diamond version of the trace modality, discuss some properties of MTL and relate it to modal and dynamic logic.

5.2 Properties of Modal Trace Logic

In this section, we examine some properties of MTL. We define a trace semantics for propositional connectors of trace descriptions and for PDL programs, and show that while standard axioms of modal logic hold (partially under conditions), most axioms of PDL are not satisfied in MTL. For instance, the trace modality does not support linearization. Furthermore, we introduce a diamond version of the (box) trace modality.

Properties of Trace-Based Validity We can connect trace descriptions (in potentially different TDLs) by propositional junctors. The semantics of $\mathscr{C}_1 \subset \mathscr{C}_2$ (read: "$\mathscr{C}_1$ implies $\mathscr{C}_2$") and $\mathscr{C}_1 \odot \mathscr{C}_2$ (read: "$\mathscr{C}_1$ and $\mathscr{C}_2$") is defined as:

$$\begin{aligned} tval(K,\sigma|\mathscr{C}_1 \subset \mathscr{C}_2) &:= \overline{tval(K,\sigma|\mathscr{C}_1)} \cup tval(K,\sigma|\mathscr{C}_2) \\ tval(K,\sigma|\mathscr{C}_1 \odot \mathscr{C}_2) &:= tval(K,\sigma|\mathscr{C}_1) \cap tval(K,\sigma|\mathscr{C}_2) \end{aligned}$$

The definitions of $\mathscr{C}_1 \oplus \mathscr{C}_2$ ("$\mathscr{C}_1$ or $\mathscr{C}_2$"), $\sim \mathscr{C}$ ("not $\mathscr{C}$"), and $\mathscr{C}_1 \equiv \mathscr{C}_2$ ("$\mathscr{C}_1$ is equivalent to $\mathscr{C}_2$) follow as usual. As MTL atoms can be arbitrary TDL constructs, programs may occur *outside modalities*. For instance, the MTL formula $p \subset \varphi$, for a program p and formula φ, is well-defined. Its semantics is $\overline{tval(K,\sigma|p)} \cup tval(K,\sigma|\varphi)$—which is the semantics of $[p \Vdash \varphi]$ (with identity abstraction).[2]

We defined the semantics of first-order postconditions and LTL formulas as sets of traces starting in state σ. This implies that such a formula *never can be universally valid* in MTL: Since $tval(K,\sigma|\varphi)$ has to cover the whole universe *Traces* of traces, but only produces traces starting in σ, not even "$\Vvdash$ true" holds. We briefly discuss motivation, alternatives and remedies:

- Expressions such as $[\varphi \Vdash_\alpha p]$ can only be valid if valuations of φ do not start in different states than p. We intended to enable such applications of the trace modality; for instance, to describe that φ is a precise summary of p.
- For reasoning about validity of φ in MTL, one can use "true $\subset \varphi$" instead. Its semantics is $\overline{tval(K,\sigma|\,\text{true})} \cup tval(K,\sigma|\varphi)$; traces not starting in σ are in $\overline{tval(K,\sigma|\,\text{true})}$. The MTL formula is valid iff φ is universally valid in the classical sense.
- Defining $K,\sigma,\beta \Vvdash \mathscr{C}$ to hold iff $tval(K,\sigma,\beta|\mathscr{C}) = Traces_\sigma$ (traces starting in σ) would be an alternative. This would yield a weaker notion of validity for the trace modality as the one in [SH19b] though, since traces added by α not starting in σ (which may exist) would not have to be considered.
- It would have been an option to define validity directly as in [SH19b] without the detour via the general semantic framework of MTL. Then, we would lose the

[2] It still makes sense to define the trace modality independently: First, because the ubiquitous approximation aspect is more obvious in the trace modality notation. And secondly, which is more important, to include *abstraction*, the second of the three principles guiding the design of MTL.

attractive uniformity of MTL; also, the meaning of expressions such as $[\mathscr{C}_1 \Vdash_\alpha [\mathscr{C}_2 \Vdash_{\alpha'} \mathscr{C}_3]]$ is not clear if the trace modality has no trace semantics.

Diamond Modality As in DL, we can define a dual (diamond) modality, with the intuition that $|\models \langle \mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec} \rangle$ holds if, for every initial state, *there is* a trace of the implementation which is also a trace of the specification. This does not permit the implementation trace set to be empty. On the other hand, not all traces of the implementation have to be approximated by the specification—one suffices. For *deterministic* programs, which always evaluate to a *single* trace for a given initial state, the diamond modality is stronger. We define the semantics of the diamond trace modality as:

$$\begin{aligned}&tval\big(K,\sigma|\langle \mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}\rangle\big) :=\\ &\qquad\qquad \big\{\tau \in Traces \mid \exists \tau';\ \tau' \in \big(\alpha\big(tval\big(K,\sigma,\beta|\mathscr{C}_{impl}\big)\big) \cap \alpha\big(tval\big(K,\sigma,\beta|\mathscr{C}_{spec}\big)\big)\big)\big\}\end{aligned}$$

Relation to Modal and Dynamic Logic We consider two important axioms of modal logic (see, e.g., [Fit07]): The *necessitation* axiom **N** (also called "modal generalization" in [HTK00]) and the *distribution* axiom **K** (also called "normality" in [Fit07]). They are important since, together with the axioms of classical logic and modus ponens, they are the axiomatic basis for "normal modal logic". Necessitation states that we can conclude $\Box\varphi$ from φ, for a modal operator $\Box$. The distribution schema is $\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$. In PDL, they are expressed as "from φ follows $[\beta]\varphi$", for a PDL program β, and "$[\beta](\varphi \to \psi) \to ([\beta]\varphi \to [\beta]\psi)$" [HTK00]. Transferring this to MTL, we show that MTL satisfies the axiom **N**. To be able to express axiom **K**, we presume an instance of MTL with the propositional implication operator "$\subset$" defined above. This axiom does not hold in general, but for relevant combinations of abstractions and TDLs. For instance, it is true for the identity abstraction, and for big-step abstraction with first-order postconditions.

For investigating whether the trace modality satisfies axioms of PDL, we embed PDL programs in MTL by giving them a trace semantics. We examine these properties from modal and dynamic logic in the remainder of this section to further characterize MTL. In Sect. 5.3, we pursue a more practical approach, expressing several program verification problems in the framework with concrete examples.

In the following, we say that an axiom schema or inference rule "holds" or "is true" if we can show its validity based on the semantics of the presumed MTL instance without additional assumptions on schematic elements in the schema or rule.

Lemma 5.2. *The trace modality satisfies axioms* **N** *of modal logic, i.e., for all trace abstractions α and TDL constructs $\mathscr{C}_1$ and $\mathscr{C}_2$, it holds that $\Vert\!\models \mathscr{C}_2$ implies $\Vert\!\models [\mathscr{C}_1 \Vdash_\alpha \mathscr{C}_2]$.*

Proof. We have to show, for any K, $\sigma \in \mathscr{S}$ and $\tau \in \mathit{Traces}$, that if $\tau \in \alpha(\mathit{tval}(K,\sigma|\mathscr{C}_1))$, it holds that $\tau \in \alpha(\mathit{tval}(K,\sigma|\mathscr{C}_2))$. From the assumption and the monotonicity of trace abstractions, we know that $\tau' \in \alpha(\mathit{tval}(K,\sigma|\mathscr{C}_2))$ for any τ', and thus also for τ. Therefore, axiom **N** is true. □

Lemma 5.3. *Let $\mathscr{C}_1$, $\mathscr{C}_2$ and $\mathscr{C}_3$ be TDL constructs, and α be a trace abstraction. Axiom* ***K*** *for the trace modality has the form*

$$[\mathscr{C}_1 \Vdash_\alpha \mathscr{C}_2 \subset \mathscr{C}_3] \subset ([\mathscr{C}_1 \Vdash_\alpha \mathscr{C}_2] \subset [\mathscr{C}_1 \Vdash_\alpha \mathscr{C}_3])$$

Axiom ***K*** *does in general* not *hold for arbitrary combinations of abstractions and TDLs. However, it is true for α_{id}, and the combination where α is α_{big} (with and without abstraction from infinite traces) and $\mathscr{C}_2$ and $\mathscr{C}_3$ are first-order postconditions.*

Proof. The schema of axiom **K** is in our framework, due to the definition of $\mathit{tval}(K,\sigma|\varphi \subset \psi)$ as $\overline{\mathit{tval}(K,\sigma|\varphi)} \cup \mathit{tval}(K,\sigma|\psi)$, equivalent to the following equation, for any K, σ:

$$\mathit{Traces} = \Bigg(\overline{\mathit{tval}\Big(K,\sigma|[\mathscr{C}_1 \Vdash_\alpha \overline{\mathit{tval}(K,\sigma|\mathscr{C}_2)} \cup \mathit{tval}(K,\sigma|\mathscr{C}_3)]\Big)} \cup \\ \Big(\overline{\mathit{tval}(K,\sigma|[\mathscr{C}_1 \Vdash_\alpha \mathscr{C}_2])} \cup \mathit{tval}(K,\sigma|[\mathscr{C}_1 \Vdash_\alpha \mathscr{C}_3])\Big)\Bigg)$$

Due to the definition of the semantics of trace modality, associativity of union, de Morgan's law, and the involution law, the right-hand side of the equation is equivalent to

$$\overline{\overline{\alpha(\mathit{tval}(K,\sigma|\mathscr{C}_1))} \cup \alpha\Big(\overline{\mathit{tval}(K,\sigma|\mathscr{C}_2)} \cup \mathit{tval}(K,\sigma|\mathscr{C}_3)\Big)} \cup \\ \overline{\overline{\alpha(\mathit{tval}(K,\sigma|\mathscr{C}_1))} \cup \alpha(\mathit{tval}(K,\sigma|\mathscr{C}_2))} \cup \overline{\alpha(\mathit{tval}(K,\sigma|\mathscr{C}_1))} \cup \alpha(\mathit{tval}(K,\sigma|\mathscr{C}_3)) = \\ \Big(\alpha(\mathit{tval}(K,\sigma|\mathscr{C}_1)) \cap \overline{\alpha\Big(\overline{\mathit{tval}(K,\sigma|\mathscr{C}_2)} \cup \mathit{tval}(K,\sigma|\mathscr{C}_3)\Big)}\Big) \cup \\ \Big(\alpha(\mathit{tval}(K,\sigma|\mathscr{C}_1)) \cap \overline{\alpha(\mathit{tval}(K,\sigma|\mathscr{C}_2))}\Big) \cup \overline{\alpha(\mathit{tval}(K,\sigma|\mathscr{C}_1))} \cup \alpha(\mathit{tval}(K,\sigma|\mathscr{C}_3))$$

We abbreviate $\mathit{tval}(K,\sigma|\mathscr{C}_1)$ with X, $\mathit{tval}(K,\sigma|\mathscr{C}_2)$ with Y and $\mathit{tval}(K,\sigma|\mathscr{C}_3)$ with Z, apply de Morgan's law, distributivity, and a complement and identity law:

$$
\begin{aligned}
&\dots \\
&= \left(\alpha(X) \cap \overline{\alpha\left(\overline{Y} \cup Z\right)}\right) \cup \left(\alpha(X) \cap \overline{\alpha(Y)}\right) \cup \overline{\alpha(X)} \cup \alpha(Z) \\
&= \left(\alpha(X) \cap \left(\overline{\alpha\left(\overline{Y} \cup Z\right)} \cup \overline{\alpha(Y)}\right)\right) \cup \overline{\alpha(X)} \cup \alpha(Z) \\
&= \left(\left(\alpha(X) \cup \overline{\alpha(X)}\right) \cap \left(\left(\overline{\alpha\left(\overline{Y} \cup Z\right)} \cup \overline{\alpha(Y)}\right) \cup \overline{\alpha(X)}\right)\right) \cup \alpha(Z) \\
&= \left(\mathit{Traces} \cap \left(\left(\overline{\alpha\left(\overline{Y} \cup Z\right)} \cup \overline{\alpha(Y)}\right) \cup \overline{\alpha(X)}\right)\right) \cup \alpha(Z) \\
&= \overline{\alpha\left(\overline{Y} \cup Z\right)} \cup \overline{\alpha(Y)} \cup \overline{\alpha(X)} \cup \alpha(Z)
\end{aligned}
$$

Since α preserves unions and after another application of de Morgan, distributivity, (in one step) the complement and identity laws, commutativity of union and de Morgan, and removal of the abbreviations for X, Y and Z, we obtain

$$
\begin{aligned}
&\overline{\alpha\left(\overline{Y} \cup Z\right)} \cup \overline{\alpha(Y)} \cup \overline{\alpha(X)} \cup \alpha(Z) \\
&= \overline{\alpha\left(\overline{Y}\right) \cup \alpha(Z)} \cup \overline{\alpha(Y)} \cup \overline{\alpha(X)} \cup \alpha(Z) \\
&= \left(\left(\overline{\alpha\left(\overline{Y}\right)} \cap \overline{\alpha(Z)}\right) \cup \alpha(Z)\right) \cup \overline{\alpha(Y)} \cup \overline{\alpha(X)} \\
&= \overline{\alpha\left(\overline{Y}\right)} \cup \alpha(Z) \cup \overline{\alpha(Y)} \cup \overline{\alpha(X)} \\
&= \overline{\alpha\left(\overline{Y}\right) \cap \alpha(Y)} \cup \alpha(Z) \cup \overline{\alpha(X)} \\
&= \overline{\alpha\left(\overline{\mathit{tval}(K, \sigma|\mathscr{C}_2)}\right) \cap \alpha(\mathit{tval}(K, \sigma|\mathscr{C}_2))} \cup \\
&\qquad \overline{\alpha(\mathit{tval}(K, \sigma|\mathscr{C}_1))} \cup \alpha(\mathit{tval}(K, \sigma|\mathscr{C}_3))
\end{aligned}
$$

Axiom **K** holds iff any trace τ is element of this set. If τ is *no* trace of the abstracted implementation $\mathscr{C}_1$ or *is* a trace of the abstracted postcondition $\mathscr{C}_3$, this is the case. Conversely, let $\tau \in \alpha(\mathit{tval}(K, \sigma|\mathscr{C}_1))$, but $\tau \notin \alpha(\mathit{tval}(K, \sigma|\mathscr{C}_3))$. Note that such a trace generally exists for postconditions which are no theorems, and abstractions that are not too coarse. Then, it is the case that axiom **K** holds if, and only if,

$$\tau \notin \alpha\left(\overline{\mathit{tval}(K, \sigma|\mathscr{C}_2)}\right) \cap \alpha(\mathit{tval}(K, \sigma|\mathscr{C}_2)). \tag{5.1}$$

A sufficient condition for this to hold is that α is also homomorphic on intersections, i.e., $\alpha(A) \cap \alpha(B) = \alpha(A \cap B)$ (since it holds that $\alpha(\emptyset) = \emptyset$). This is not generally necessary and does not hold for many sensible abstractions. For example, if p and q are *different* programs with *the same final states*, it holds that

$$\alpha_{big}(tval(K,\sigma|p)) \cap \alpha_{big}(tval(K,\sigma|q)) = \alpha_{big}(tval(K,\sigma|p)) = \alpha_{big}(tval(K,\sigma|q)),$$

while $\alpha_{big}(tval(K,\sigma|p) \cap tval(K,\sigma|q)) = \alpha_{big}(\emptyset) = \emptyset$.

Nonetheless, for many combinations of abstractions and TDLs, axiom **K** *does* hold. In particular, Eq. (5.1) is always true for $\alpha = \alpha_{id}$. The combination of big-step abstraction α_{big} and first-order postconditions for $\mathscr{C}_2$ and $\mathscr{C}_3$ behaves well, too, in both variants α_{big}^{inf} and α_{big}^{fin}. In the latter case, the abstraction does not add traces to the semantics of first-order formulas $\mathscr{C}_2$ (thus, Eq. (5.1) holds). In the former case, it adds infinite traces, but those will then also be in $\alpha_{big}(tval(K,\sigma|\mathscr{C}_3))$. □

Lem. 5.3 shows that even standard properties of MTL highly depend on the used abstractions and TDLs. Indeed, the trace modality defines a *family* of modalities for TDLs and trace abstraction, which may or may not enjoy standard properties.

To examine the properties of (propositional) DL w.r.t. the trace modality, we embed the language of regular abstract DL programs [HTK00]. A DL program consists of *atomic* programs $a, b, c, \ldots$ and the following program operators, where β, γ are programs:

	Name	Example	Intuition
;	composition	$\beta;\gamma$	"Execute β, then execute γ."
$\cup$	choice	$\beta \cup \gamma$	"Choose β or γ nondeterministically and execute it."
$*$	iteration	$\beta*$	"Execute β a nondeterministically chosen finite number of times (zero or more)."
?	test	$\varphi?$	"Test φ; proceed if true, fail if false."

Following [HTK00], we define the semantics of compound PDL programs as:

$$\begin{aligned} tval(K,\sigma|\beta;\gamma) &:= \{\sigma_1 \cdots \sigma_i \cdots \sigma_n \mid \sigma_1 \cdots \sigma_i \in tval(K,\sigma|\beta) \text{ and} \\ &\qquad\qquad \sigma_i \cdots \sigma_n \in tval(K,\sigma|\gamma)\} \\ tval(K,\sigma|\beta \cup \gamma) &:= tval(K,\sigma|\beta) \cup tval(K,\sigma|\gamma) \end{aligned}$$

$$
\begin{aligned}
tval(K,\sigma|\beta*) &:= \{\sigma\sigma \mid \sigma \in \mathscr{S}\} \cup tval(K,\sigma|\beta) \cup tval(K,\sigma|\beta;\beta) \cup \ldots \\
tval(K,\sigma|\varphi?) &:= \{\sigma\sigma \mid K,\sigma \models \varphi\}
\end{aligned}
$$

The following Lem. 5.4 establishes that only one axiom of the axiomatic system of PDL [HTK00, Sect. 5.5] generally holds for the trace modality (the axiom (iv) for choice); the axiom for a conjunctive postcondition, axiom (iii), holds without abstraction. Axiom (ii) of the system of is axiom **K** which was already discussed in Lem. 5.3, item (i) are the standard axioms of propositional logic. All other axioms do not hold in MTL. For instance, the linearization axiom $[\beta;\gamma \Vdash_\alpha \varphi] \leftrightarrow [\beta \Vdash_\alpha [\gamma \Vdash_\alpha \varphi]]$ cannot be shown, since the trace modality evaluates as well the implementation as the specification starting in the same state, whereas in $\beta;\gamma$, program γ is started in the final state of β. That these axioms do not characterize MTL is not surprising: PDL is a system for postcondition reasoning, while MTL is a much more flexible framework for trace inclusion.[3] Expressing relational properties in a dynamic logic requires relatively sophisticated formalizations, e.g., with uninterpreted predicates to collect effects (see Sect. 6.1 for a formalization in JavaDL). Moreover, in a big-step system like PDL, finer granularities are impossible to convey. An alternative to linearization more suitable for the trace modality is *symbolic evaluation* of the implementation and specification; the resulting *symbolic* traces then can be checked for inclusion. The approach proposed in Sect. 5.4 follows this direction.

Lemma 5.4 (Axioms of PDL). *Axiom (iii) of PDL holds for the trace modality with identity abstraction. Axiom (iv) is generally true; axioms (v) to (viii) do not hold.*

Proof. See Appendix D. □

5.3 Formalization of Verification Tasks

In the following, we describe several common program verification tasks and recommend formalizations in Modal Trace Logic. This demonstrates how the building blocks of MTL, i.e., different Trace Description Languages and trace abstractions, can be combined to express verification problems in a uniform way. Table 5.1 (Page 194) shows a condensed summary of our formalizations.

[3] Note that also JavaDL does not support linearization, due to the pre- and postfixes π and ω.

5.3.1 Functional Verification

In functional verification, one shows a program p to satisfy a postcondition *Post* provided that a precondition *Pre* holds initially. The problem is frequently formalized with *Hoare triples* $\{Pre\}\, p\, \{Post\}$ [Hoa69]. In DL [Ahr+16; HTK00], this corresponds to $Pre \rightarrow [p]Post$. We distinguish *partial correctness*, where *Post* is asserted to hold *if* p terminates (or, more strictly, completes normally), from *total correctness*, where it is also shown *that* p terminates (completes normally). For the latter one can use the dual modality $\langle p \rangle Post$.

Functional correctness is over the TDLs of programs (for the implementation) and first-order postconditions (for the specification). We use big-step abstraction with abstraction of infinite traces α^{inf}_{big} for partial functional correctness, resulting in

$$Pre^{ltl} \subset [p \Vdash_{\alpha^{inf}_{big}} Post]$$

We use Pre^{ltl} instead of *Pre*, since our first-order formulas have a *post*condition semantics. As we defined the LTL counterpart (Sect. 5.1) to represent all traces *starting* with states satisfying *Pre*, it is suitable for use as a precondition. Note that this peculiarity is due to our specific formalization of first-order logic; different ones are possible, as the only *mandatory* element with a *fixed semantics* of MTL is the trace modality.

For total correctness, we use α^{fin}_{big}. In a standard program trace semantics, using this abstraction alone is sufficient for reasoning about *termination:* All traces of the implementation, including the infinite ones, have to be traces of the specification. If the specification construct does not directly represent infinite traces, as in the case of first-order postconditions, $[p \Vdash_{\alpha^{fin}_{big}} Post]$ cannot be proven. Nonetheless, we recommend the diamond modality for total correctness (of *deterministic* programs). If failed assertions or uncaught exceptions are semantically represented as an empty trace set, the box version would trivially hold. Therefore, total functional correctness is formalized in MTL as

$$Pre^{ltl} \subset \langle p \Vdash_{\alpha^{fin}_{big}} Post \rangle$$

Example 5.1. Let $p :=$`i++; j=i*i; while (i<=j) { i=i*2; }`. The program diverges iff the initial value of `i` is negative. One can prove postcondition *even*(`i`) (with the obvious meaning) for p *if it terminates* (partial correctness). Thus, $K, \sigma \mid\mid\!\!= [p \Vdash_{\alpha^{inf}_{big}} even(i)]$ must hold in all K, σ, i.e., $Traces = \overline{\alpha^{inf}_{big}(tval(K, \sigma|p))} \cup \alpha^{inf}_{big}(tval(K, \sigma|even(i)))$. If $\sigma(\mathtt{i}) < 0$, then $tval(K, \sigma|p)$ contains a single infinite trace. Since it is infinite, this trace is also contained in $\alpha_{big}^{inf}(tval(K,\sigma|even(i)))$. If $\sigma(\mathtt{i}) \geq 0$, $tval(K,\sigma|p)$ has a single finite trace whose final state assigns an even value to $\mathtt{i}$ (because $\sigma(\mathtt{i})+1$ was multiplied by 2 a number of times in the loop's body). Hence, $\alpha_{big}^{inf}(tval(K,\sigma|p))$ contains (all infinite traces and) all finite traces stating in σ and ending in a state satisfying $even_{\mathtt{i}}$. These traces are in $\alpha_{big}^{inf}(tval(K,\sigma|even(i)))$ by definition of the trace semantics for first-order postconditions.

We cannot show the condition for total correctness, $K,\sigma \Vert\!\models \langle p \Vdash_{\alpha_{big}^{fin}} even(\mathtt{i})\rangle$, for any σ with $\sigma(\mathtt{i}) < 0$, because then there is an infinite trace in $\alpha_{big}^{fin}(tval(K,\sigma|p))$ which is not in $\alpha_{big}^{fin}(tval(K,\sigma|even(\mathtt{i})))$. However, $\Vert\!\models \mathtt{i} \geq 0^{ltl} \subset \langle p \Vdash_{\alpha_{big}^{fin}} even(\mathtt{i})\rangle$ is true:

$$tval\Big(K,\sigma|\mathtt{i} \geq 0^{ltl} \subset \langle p \Vdash_{\alpha_{big}^{fin}} even(\mathtt{i})\rangle\Big) = \\ \overline{tval\big(K,\sigma|\mathtt{i} \geq 0^{ltl}\big)} \cup \overline{\alpha_{big}^{fin}(tval(K,\sigma|p))} \cup \alpha_{big}^{fin}(tval(K,\sigma|even(i)))$$

is equivalent to *Traces* since all traces starting with states where $\mathtt{i}$ is negative are included in the complement of $tval\big(K,\sigma|\mathtt{i} \geq 0^{ltl}\big)$.

Normal completion (and thus, also termination) only is, for *deterministic* programs, expressed as $\langle p \Vdash_{\alpha_{big}^{fin}} \mathrm{true}\rangle$: Since $\alpha_{big}^{fin}(tval(K,\sigma|\,\mathrm{true}))$ contains all *finite* traces, all traces of p have to be finite. For nondeterministic programs, we still can reason about *termination* using $[p \Vdash_{\alpha_{big}^{fin}} \mathrm{true}]$; this formalization, however, allows p to complete abruptly, that is, to evaluate to the empty trace set. The diamond version is not useful for nondeterministic programs since only *one* trace starting in σ has to terminate (others may diverge). $\Diamond$

5.3.2 Information Flow Analysis

To prove that a given program treats secret inputs (for example, a password) confidentially, i.e. it does not inadvertently leak secret information, one can formally prove that it satisfies an *information flow policy*. In the simplest case such policies partition program variables into *low*-security variables that hold observable values and *high*-security ones whose values are secret. A policy imposes restrictions on the flow of values from *high* to *low* variables. A standard and very strong policy is *non-interference*: “Whenever two instances of the same program are run with equal *low* values and arbitrary *high* values, then the resulting *low* values are equal in the final state”. This ensures that an attacker cannot learn anything about secret values by running the program with observable values. For simplicity, assume a program p contains exactly one low variable $\mathtt{l}$ and one high variable $\mathtt{h}$, written $p(\mathtt{l},\mathtt{h})$.

Using *self composition* [BDR04; DHS05], this is formalized as a Hoare triple: If we can prove $\{\mathtt{l} \doteq \mathtt{l}'\}\, p(\mathtt{l},\mathsf{h}); p(\mathtt{l}',\mathsf{h}')\, \{\mathtt{l} \doteq \mathtt{l}'\}$, p satisfies non-interference. It can also be directly expressed with the trace modality: $\Vert\!\models [p(\mathtt{l},\mathsf{h}) \Vdash_{\alpha^{fin}_{big} \circ \alpha_{\{\mathtt{l}\}}} p(\mathtt{l},\mathsf{h}')]$. Note that the renaming of $\mathtt{l}$ to $\mathtt{l}'$ is then not necessary since programs are not composed, but evaluated separately.

Example 5.2. Let $p :=$`l=42; if (h>20) {l=17;}`. This program does not satisfy non-interference, because the final value of the observable variable `l` depends on the initial value of `h`. We prove that indeed, $\Vert\!\models [p(\mathtt{l},\mathsf{h}) \Vdash_{\alpha^{fin}_{big} \circ \alpha_{\{\mathtt{l}\}}} p(\mathtt{l},\mathsf{h}')]$ does *not* hold, by showing that there is a state $\sigma \in \mathscr{S}$ for which

$$(\alpha^{fin}_{big} \circ \alpha_{\{\mathtt{l}\}})(tval(K, \sigma | p(\mathtt{l},\mathsf{h}))) \subseteq (\alpha^{fin}_{big} \circ \alpha_{\{\mathtt{l}\}})(tval\big(K, \sigma | p(\mathtt{l},\mathsf{h}')\big))$$

is not true. Let σ be such that $\sigma(\mathtt{l}) = 0$, $\sigma(\mathsf{h}) = 0$ and $\sigma(\mathsf{h}') = 30$. Then the trace set of the implementation is of the form

$$\{(\{\mathtt{l} \mapsto 0, \mathsf{h} \mapsto \dots\} \cdots \{\mathtt{l} \mapsto 42, \mathsf{h} \mapsto \dots\})\}$$

which is not contained in the set for the specification

$$\{(\{\mathtt{l} \mapsto 0, \mathsf{h} \mapsto \dots\} \cdots \{\mathtt{l} \mapsto 17, \mathsf{h} \mapsto \dots\})\}. \qquad \Diamond$$

Declassification, such as *delimited information release* [SM03], can be easily encoded via preconditions. Let e be an expression we want to declassify. We assume the existence of expressions **`declassify`**`(`e`)`, as in [SM03], which evaluate to e while permitting flow of e to the *low* level. As for programs, write $e(\mathtt{l},\mathsf{h})$ to make the variables occurring in e explicit. Then non-interference with declassification is formalized as:

$$\Vert\!\models (e(\mathtt{l},\mathsf{h}) \doteq e(\mathtt{l},\mathsf{h}'))^{ltl} \subset [p(\mathtt{l},\mathsf{h}) \Vdash_{\alpha_{\{\mathtt{l}\}} \circ \alpha_{big}} p(\mathtt{l},\mathsf{h}')] \ .$$

Example 5.3 (Declassification)**.** We consider the classic PIN example, where a *low* variable `OK` is set to `true` depending on whether a *high* input `inp` equals a *high* variable `pin` containing a PIN. Let

$p :=$ **`if`** `(`**`declassify`**`(pin==inp)) { OK=`**`true`**`; }` **`else`** `{ OK=`**`false`**`; }`

be this program. If we do not give special semantics to the **`declassify`** expression, there is an initial state σ where $\sigma(\mathtt{pin}) = \sigma(\mathtt{inp})$, but $\sigma(\mathtt{pin}') \neq \sigma(\mathtt{inp})$; for this state, trace set inclusion does not hold, which is why p would be classified as insecure. The additional precondition $\left(\mathtt{pin} \doteq \mathtt{inp} \leftrightarrow \mathtt{pin}' \doteq \mathtt{inp}\right)^{ltl}$, however, rules this choice out, and we can classify the program as secure w.r.t. the delimited release semantics. ◊

5.3.3 Software Model Checking

Software Model Checking (SMC) [JM09] describes a wide range of techniques for analyzing *safety* or *liveness* properties of programs. Those techniques have in common that they focus on *automation* (usually, full automation while requiring limited or no user input) at cost of expressivity. Frequently, the goal is not to prove correctness relative to a specification, but to quickly uncover bugs or to generate high-coverage test cases. Recently, there has been a *convergence* between *model checking* and *deductive verification* techniques [Sha18], as more mechanisms traditionally known from the latter field, such as abstraction [Vis+03], symbolic execution [PV04], etc., are integrated to achieve greater expressivity. On the other side, Bounded Model Checking (BMC) approaches [CKL04], which limit state space exploration by a user-defined upper bound on loop unwindings, are well-known and successful, and finite space checkers such as SPIN [Hol97] continue being used, e.g. in protocol verification. Properties of interest to SMC (e.g., the absence of memory faults) can usually be formalized in Temporal Logic (TL). In the following, we formalize four popular SMC approaches in MTL. Formulas ξ are TL formulas.

Finite Space MC Finite space model checkers (SPIN [Hol97] is a prominent representative) exhaustively explore the state space of an abstract program model. This implies that the analysis starts from *a concrete input state* σ and that no unbounded data structures are involved. We can formalize this problem as $\sigma \mid\!\models [p \Vdash \xi]$.

Bounded MC (BMC) BMC [Bie+03; CKL04] handles unbounded data structures, but restricts the search space according to a predefined upper bound on the number of loop executions. This problem can be expressed as $\mid\!\models [p \Vdash_{\alpha_p^k} \xi]$, where α_p^k adds all traces of p with more than k loop executions[4] (and recursive method calls).

Abstraction-Based MC uses data abstraction to limit the search space. We express it as $\mid\!\models [p \Vdash_{\alpha_d} \xi]$, where α_d is an abstract interpretation of the data types of p.

Symbolic Execution-Based MC This variant of SMC is similar to functional verification (Sect. 5.3.1). They mainly differ in the used abstraction (identity vs. big-step) and that in MC less complex properties are proven: $\mid\!\models [p \Vdash \xi]$.

Model Checking tools for bug finding can be formalized with the diamond trace modality: They eagerly try to show $\mid\!\models \langle p \Vdash \neg\xi \rangle$, i.e. there is a trace of p violating ξ. Such a counterexample can be used to fix the program, and/or to create a useful test case.

So far, we considered *concrete* programs. The two subsequently discussed verification tasks are over *schematic* (abstract) programs.

5.3.4 Program Synthesis

Automated program synthesis starts with a specification of programs at a higher level than executable code. The latter is created (semi-)automatically from the specification. In [SGF10], for instance, the user supplies a *scaffold* consisting of a functional specification (*Pre*, *Post*), domain constraints defining the domains of expressions and guards, and a *schematic program* (called "flowgraph template") of the form $\bullet | *(T)|T;T$. Here, $\bullet$ is an acyclic fragment, T again a schematic program and $*(T)$ a loop with body T. The synthesizer infers *synthesis conditions*. These are satisfiable whenever there exists a valid program for the scaffold.

We encode $\bullet$ of the flowgraph template by programs p with schematic statements `P`, `Q`, etc. Synthesis conditions are included in the intermediate program as suitable **`assert`**(φ) statements. When refining an intermediate program p to a more concrete program p', the property to show is $\|\models \mathit{Pre}^{ltl} \subset [p' \Vdash_{\alpha^{fin}_{big}} p]$: that p' is indeed a refinement of p. When replacing a schematic statement `P` with a concrete one, we also have to remove the synthesis condition for `P` since otherwise, p' might yield the empty set of traces and trivially satisfy that property. Alternatively, we could use the diamond modality (when synthesizing deterministic programs).

Example 5.4 discusses our formalization of the program synthesis problem within the trace modality along an example from the literature computing integer square roots.

Example 5.4. We consider the square root example from [SGF10]. Given a user-defined specification $\mathit{Pre} := \mathtt{x} \geq 1$, $\mathit{Post} := \mathtt{i}^2 \leq \mathtt{x} < (\mathtt{i}+1)^2$ and scaffold program $\bullet;*(\bullet);\bullet$, the synthesizer should generate a program `IntSqrt(`**`int`**` x)` satisfying the specification (i.e., computing the integer square root of a strictly positive variable `x`) and matching the structure of the scaffold. An additional user-defined constraint is that, apart from `x` and `i`, there must be at most one additional variable `v`, also of integer type. Listing 5.1 shows a concrete program matching the specification. To apply our formalization, we first translate the scaffold in a schematic program: `P;`**`while`**` (b){Q};R`. Let now $\mathit{Syn}_{\mathtt{P/Q/R}}$ be synthesis conditions for `P`, `Q` and `R` inferred by the synthesizer. Note that $\mathit{Syn}_{\mathtt{Q}}$ is an inductive invariant for `Q`. A concrete instantiation for $\mathit{Syn}_{\mathtt{Q}}$ is $\mathtt{v} \doteq \mathtt{i}^2 \land \mathtt{x} \geq (\mathtt{i}-1)^2 \land \mathtt{i} \geq 1$. The scaffold annotated by **`assert`** statements for the synthesis conditions is depicted in

Listing 5.1: `IntSqrt`

```
v = 1; i = 1;

while (v<=x) {
  v = v+2i+1;
  i++;
}
i = i-1;
```

Listing 5.2: Annotated scaffold for `IntSqrt`

```
P;
assert(v ≐ 1 ∧ i ≐ 1 ∧ x ≐ x′);
while (v<=x) {
  Q;
  assert(v ≐ i² ∧ x ≥ (i − 1)² ∧ i ≥ 1);
}
R;
assert(i² ≤ x < (i + 1)²);
```

Listing 5.2 (we write $\mathtt{x}'$ for the value of `x` before the execution of a schematic statement). Suppose that now we refine the scaffold *sc* to a program p by replacing `Q` *and* the following **`assert`** statement by the subsequent program q: `v=v+2i+1;i++;`. To prove this correct, we have to show $\Vdash \mathtt{i{\geq}1}^{ltl} \subset [p \Vdash_{\alpha^{fin}_{big}} sc]$. Since the traces for *sc* include one trace for each possible instantiation of `Q` satisfying $Syn_{\mathtt{Q}}$ in the given context, and q also satisfies $Syn_{\mathtt{Q}}$ in this context, this is true. We point out that we cannot instead show $Pre^{ltl} \subset [q \Vdash_{\alpha^{fin}_{big}} Syn_{\mathtt{Q}}]$, since the program before the insertion position, i.e. already substituted concrete programs as well as asserted synthesis conditions, also has to be considered. Here, in particular, it is important that `v` and `i` initially are 1 for the invariant to hold. ◊

5.3.5 Correct Compilation

A compiler translates a program p of a source language into a program c of a target language, preserving the behavior of p. The translation can introduce new program variables. Then, preservation of behavior is typically restricted to a set of *observable* variables *obs*. In *modular* compilation, a program p is given within an unspecified context. In this case, both p and c are *abstract*. Correctness of compilation can be expressed as $\Vdash [c \Vdash_{\alpha_{obs} \circ \alpha^{fin}_{big}} p]$. If we want to enforce inclusion of the traces of c in the traces of p, we can—for deterministic languages—use the diamond modality instead. In particular for non-deterministic languages, we can *additionally* prove the reverse direction $\Vdash [p \Vdash_{\alpha_{obs} \circ \alpha^{fin}_{big}} c]$.

The formalization makes the similarity to program synthesis explicit. Indeed, one could create a scaffold by extracting synthesis conditions from p, and then try to infer c automatically. For example, in [SH18], a Symbolic Execution Tree of the source program is "mined" to extract the target program. It is related to proof mining techniques used in program synthesis. A popular approach for correct compilation is the specification of the compiler within the executable fragment of an interactive proof assistant like Isabelle or Coq, as in CompCert [Ler09]. Reference [SH18] proposes a rule-based technique using *simultaneous* Symbolic Execution (SE) with a *dual modality*, which can be seen as a specialization of the trace modality. The interesting property of this framework is that compilation rules can be proven automatically based on SE calculi for the source and target language. We think that a similar technique might be applicable to other verification tasks.

5.3.6 Program Evolution & Bug Fixing

Sometimes, the behavior of the specification should intentionally be *not* preserved. This situation occurs in program evolution, e.g., after manual or automatic bug fixing [Mon18]: the patched program is supposed to exhibit the bug no longer, but no new bug is to be introduced. Similarly, in fault propagation analysis, an injected fault typically *will* change the behavior of a program, but not arbitrarily. This problem is most naturally expressed as $\Vdash [p_{fixed} \Vdash_{\alpha_{patch} \circ \alpha_{big}^{fin}} p_{buggy}]$, where the "patch abstraction" α_{patch} adds traces describing the fix. This way, it can be ensured that in p_{fixed}, no new behavior is added w.r.t. p_{bug} apart from the fix. Furthermore, this abstraction can be seen as a *documentation* of the patch. Additionally, the bug itself can be encoded into a "bug" trace abstraction α_{bug} and used in a reverse proof obligation $\Vdash [p_{buggy} \Vdash_{\alpha_{bug} \circ \alpha_{big}^{fin}} p_{fixed}]$ to demonstrate that no behavior is lost. The bug abstraction could also be obtained from a counter example trace. As usual, the diamond trace modality is an obvious alternative for deterministic programs.

Example 5.5. We explain the mentioned techniques along a simple example. The program $p_{buggy} := \texttt{\textbf{if} (x<-1) \{x=-x;\}}$ should compute the absolute of a given integer x; i.e., after execution of the program, x should be positive. However, the program contains a bug: The programmer misspelled the "<" operator which should be a "<=" instead. For the input -1, a wrong result is produced. Let p_{fixed} be the corrected program. The trace set $\mathcal{T}_{patch}$ describes the new traces added by the patch:

$$\mathcal{T}_{patch} := \{\sigma\sigma[\mathtt{x} \mapsto -\sigma(\mathtt{x})] \mid \sigma \in \mathcal{S} \wedge \sigma(\mathtt{x}) = -1\}$$

The dual trace set describing the buggy traces, $\mathcal{T}_{bug}$, is defined as

$$\mathcal{T}_{bug} := \{\sigma\sigma \mid \sigma \in \mathcal{S} \wedge \sigma(\mathtt{x}) = -1\}$$

The path and bug abstractions are consequently defined as $\alpha_{patch}(\mathcal{T}) := \mathcal{T} \cup \mathcal{T}_{patch}$ and

$\alpha_{bug}(\mathcal{T}) := \mathcal{T} \cup \mathcal{T}_{bug}$. Then, both $\Vdash \langle p_{fixed} \Vdash_{\alpha_{patch} \circ \alpha_{big}^{fin}} p_{buggy} \rangle$ and $\Vdash \langle p_{buggy} \Vdash_{\alpha_{bug} \circ \alpha_{big}^{fin}} p_{fixed} \rangle$ can be proven. We use the diamond modality since our programs are deterministic. $\Diamond$

Table 5.1: Representations of Different Verification Tasks in MTL

Task	Problem	Solution Techniques (excerpt)
Partial Correctness	$\Vdash [p \Vdash_{\alpha_{big}^{inf}} Post]$	Symbolic execution, weakest precondition reasoning, Hoare calculus
Total Correctness	$\Vdash \langle p \Vdash_{\alpha_{big}^{fin}} Post \rangle$	ditto; plus reasoning about variant / ranking function
Information Flow	$\Vdash [p(\mathtt{l}, \mathtt{h}) \Vdash_{\alpha_{\{l\}} \circ \alpha_{big}^{fin}} p(\mathtt{l}, \mathtt{h}')]$	Security type systems, Hoare calculus, symbolic execution
Information Flow with Declassification	$\Vdash \bigwedge_{i=1}^{n} (e_i(\mathtt{l}, \mathtt{h}) \doteq e_i(\mathtt{l}, \mathtt{h}')) \subset [p(\mathtt{l}, \mathtt{h}) \Vdash_{\alpha_{\{l\}} \circ \alpha_{big}^{fin}} p(\mathtt{l}, \mathtt{h}')]$	ditto
Finite Space MC	$\sigma \Vdash [p \Vdash \xi]$	Automata constructions
Bounded MC	$\Vdash [p \Vdash_{\alpha_p^k} \xi]$	SMT solvers for checking encoded program paths
Abstraction-Based MC	$\Vdash [p \Vdash_{\alpha_d} \xi]$	Overapproximation techniques, CEGAR loops
Symbolic Execution-Based MC	$\Vdash [p \Vdash \xi]$	Invariant generation, k-induction
Bug Finding	$\Vdash \langle p \Vdash \neg\xi \rangle$	All MC techniques; can be integrated with all abstractions
Program Synthesis	$\Vdash [p' \Vdash_{\alpha_{big}^{fin}} p]$	Proof-theoretic synthesis, proof mining
Correct compilation	$\Vdash [c \Vdash_{\alpha_{obs} \circ \alpha_{big}^{fin}} p]$	Simultaneous symbolic execution, compiler extraction from executable HOL specifications
Program evolution / Bug fixing	$\Vdash [p_{fixed} \Vdash_{\alpha_{patch} \circ \alpha_{big}^{fin}} p_{buggy}]$ $\Vdash [p_{buggy} \Vdash_{\alpha_{bug} \circ \alpha_{big}^{fin}} p_{fixed}]$	Manual program refinement, automatic software repair

5.4 Symbolic Trace Logic

Modal Trace Logic is a logic with only one mandatory syntactic construct: the trace modality. It is a “plugin” logic where additional syntax elements (Trace Description Languages) can be added as long as they have a trace semantics. Therefore, it is not possible to construct a calculus working on the concrete syntax of MTL, since even the trace modality is parametric in the TDLs of implementation and specification, and in the trace abstraction. One *could* devise calculi for different concrete choices of TDLs and trace abstractions. For instance, the JavaDL calculus could be used for an instantiation of MTL to Java programs as "implementation" TDL, first-order postconditions as "specification" TDL, big-step abstraction (α_{big}^{inf} for the JavaDL box modality and α_{big}^{fin} for diamond) and formulas with *pre*condition semantics (φ^{ltl} in the terminology of Sect. 5.1) outside modalities. Instead of investigating such specialized calculi, in this chapter we propose Symbolic Trace Logic (STL), a logic operating on *regular symbolic traces*. STL is an independent logic, with its own syntax (symbolic traces), semantics (sets of concrete traces) and calculus (a sequent calculus operating on symbolic traces). However, since STL symbolic traces and MTL expressions both represent sets of execution traces, it is possible to approximate MTL expressions by symbolic traces and reason about their validity using the STL calculus. Thus, when integrating a TDL into MTL, one obtains an MTL-based logic *and* a sound reasoning system by not only defining a *trace-based semantics* of the TDL, but also a *translation to symbolic traces*.

Our symbolic trace language resembles regular programs of PDL (which are named regular since their syntax resembles regular expressions); the "atomic programs" are Symbolic Execution States (SESs). Each symbolic trace ϖ represents a set of concrete traces $[\![\varpi]\!]$.[5] Thus, we can represent, for instance, the criterion for the validity of a trace modality formula $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ as a symbolic trace expression $\neg\mathfrak{a}(\varpi_{impl}) + \mathfrak{a}(\varpi_{spec})$, where ϖ_{impl} and ϖ_{spec} are symbolic traces representing the implementation and specification constructs, $\mathfrak{a}$ is a symbolic version of α and $\neg$, $+$ symbolically represent set complement and union. If the symbolic traces and $\mathfrak{a}$ are *precise*, it holds that

$$tval\big(K, \sigma | \neg\mathfrak{a}(\varpi_{impl}) + \mathfrak{a}(\varpi_{spec})\big) = tval\big(K, \sigma | [\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]\big)\,.$$

A sequent in the STL calculus has the form $\varpi_1, \ldots, \varpi_n \vdash \varpi_1, \ldots, \varpi_m$. As usual, the traces $\varpi_1 \ldots, \varpi_n$ are called the antecedents of the sequent; the traces $\varpi_1, \ldots, \varpi_m$ on the right side of the sequent separator $\vdash$ are called the succedents. Both the antecedent and succedent are sets, i.e., order and multiple occurrences are not relevant. The meaning of an STL sequent is $\bigcup_{1=i}^{n}[\![\varpi_i]\!] \cup \overline{\bigcap_{1=j}^{m}[\![\varpi_j]\!]}$. An STL sequent *seq* is *valid* iff $[\![seq]\!] = Traces$. Consequently, the judgment $\vdash \varpi$ expresses that $[\![\varpi]\!] = \emptyset$, or conversely, $[\![\neg\varpi]\!] = Traces$.

We exemplarily demonstrate this idea for a trace modality formula. Let, as above, ϖ_{impl} and ϖ_{spec} be symbolic traces for the implementation and specification parts of a formula $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$; we assume that $\mathfrak{a}$ has already been applied and, for now,

[5] We sometimes write "$[\![\circ]\!]$" as an informal shorthand notation for "$tval(K, \sigma|\circ)$, for all K, σ".

that symbolic trace extraction and abstraction are precise, s.t., e.g., $tval\big(K,\sigma|\varpi_{impl}\big) = tval\big(K,\sigma|\alpha\big(\mathscr{C}_{impl}\big)\big)$. To prove $|\!\models [\mathscr{C}_{impl} \Vdash_{\alpha} \mathscr{C}_{spec}]$, we create a sequent $\vdash \neg\big(\neg\varpi_{impl} + \varpi_{spec}\big)$, which we simplify by a series of calculus rule applications as follows:

$$\dfrac{\dfrac{\dfrac{\varpi_{spec} \;\vdash\; \varpi_{impl}}{\neg\varpi_{impl},\, \varpi_{spec} \;\vdash}\;\neg\text{left}}{\neg\varpi_{impl} + \varpi_{spec} \;\vdash}\;+\text{left}}{\vdash\; \neg\big(\neg\varpi_{impl} + \varpi_{spec}\big)}\;\neg\text{right}$$

The resulting sequent $\varpi_{spec} \vdash \varpi_{impl}$ is valid iff $tval\big(K,\sigma|\varpi_{spec}\big) \cup \overline{tval\big(K,\sigma|\varpi_{impl}\big)} = Traces$ (for all K, σ), which is equivalent to the definition of validity of the trace modality. Expressing this equivalently as a subset relation helps to establish an intuition about STL sequents: $\varpi_{spec} \vdash \varpi_{impl}$ is valid iff $tval\big(K,\sigma|\varpi_{spec}\big) \supseteq tval\big(K,\sigma|\varpi_{impl}\big)$, i.e., we have to establish that the antecedent is "more general" than the succedent. With multiple antecedents and succedents, this amounts to

$$tval(K,\sigma|\varpi_1) \cup \cdots \cup tval(K,\sigma|\varpi_n) \supseteq tval(K,\sigma|\varpi_1) \cap \cdots \cap tval(K,\sigma|\varpi_m)$$

Thus, the sequent is valid if (but not only if) one of the following equalities holds:

(1) $tval(K,\sigma|\varpi_i) = Traces$ (for $i \in 1,\ldots,n$)

(2) $tval\big(K,\sigma|\varpi_j\big) = \emptyset$ (for $j \in 1,\ldots,m$)

(3) $tval(K,\sigma|\varpi_i) = tval\big(K,\sigma|\varpi_j\big)$ (for $i \in 1,\ldots,n$ and $j \in 1,\ldots,m$)

Rules of the STL calculus specialize sequents by equivalence-preserving transformations, (incomplete) specializations of sequences and (complete) specializations of premises to establish one of the situations "$\Gamma, \top^{\infty} \vdash \Delta$", "$\Gamma \vdash s_{\perp}, \Delta$" or "$\Gamma, \varpi \vdash \varpi, \Delta$", where $\top^{\infty}$ is the "universal" and $s_{\perp}$ an unsatisfiable symbolic state, which allows to close a branch with a closing rule. As usual, there are rules which split a proof branch into sub branches, and a closed proof is a proof with only closed branches. The soundness criterion is that a proof of a root sequent with only closed branches proves the validity of the sequent.

The calculus comprises rules with a set-theoretic foundation (as in the example above). More interesting are the specialization rules. E.g., we can specialize a premise $\top^{\infty};\varpi$ (read "any trace followed by the traces described by ϖ") to any symbolic trace $\varpi';\varpi$. For symbolic states s, s', we can transform the problem $\Gamma,(s;\varpi) \vdash \big(s';\varpi'\big),\Delta$ to $\varpi \vdash \varpi'$ if we can show that s is more general, or *subsumes*, s', written $s \rhd s'$.

We formally define the notions introduced above, starting with symbolic traces.

5.4.1 Symbolic Traces and Translations

Definition 5.6 (Symbolic Traces)**.** The set *STraces* of *Symbolic Traces* is constructed according to the following grammar, where $(C, \mathscr{U})$ is a symbolic state without program counter:

$$\begin{aligned} \mathit{STraces} ::= {} & \bot \mid \top \mid \top^{\infty} \mid (C, \mathscr{U}) \mid \\ & \neg \mathit{STraces} \mid \mathit{STraces}; \mathit{STraces} \mid \mathit{STraces} + \mathit{STraces} \mid \mathit{STraces}^{*} \mid \mathit{STraces}^{\omega} \end{aligned}$$

Atomic symbolic traces are the *empty trace* $\bot$, the *universal finite trace* $\top$ representing all finite traces, the *universal trace* $\top^{\infty}$ representing the *whole* universe of traces, and symbolic states $(C, \mathscr{U})$ for singleton traces. As in Chapter 3, the path condition C is a set of closed formulas, and the symbolic store $\mathscr{U}$ an update. If ϖ and ϖ' are symbolic traces, then $\neg\varpi$ is the *complement* of ϖ, $\varpi; \varpi'$ is the *sequential composition* of ϖ and ϖ', $\varpi + \varpi'$ models nondeterministic *choice*, ϖ^{*} *finite looping*, and ϖ^{ω} *potentially infinite looping*. ◊

Notation. We write (C) for SESs (C, Skip) with empty symbolic store. Similarly, we omit the trivial path condition $\{\text{true}\}$ (which is equivalent to $\emptyset$), writing $(\mathscr{U})$ instead of $(\emptyset, \mathscr{U})$ and $(\{\text{true}\}, \mathscr{U})$. Furthermore, we write φ instead of $\{\varphi\}$ for singleton path conditions. ◊

We define the semantics of symbolic traces via the trace valuation function *tval*. The semantics for singleton symbolic traces $(C, \mathscr{U})$ is its *concretization* to concrete states (cf. Sect. 3.1), which, for a fixed structure K and concrete state $\sigma \in \mathscr{S}$, is either an empty set (if the path condition is not satisfied) or the singleton set resulting from transforming σ according to the symbolic store. There is one speciality: "Free" program variables in path conditions are treated universally. For instance, $\mathit{tval}(K, \sigma|(\mathtt{x} > 0))$ always evaluates to the set of all states where $\mathtt{x}$ is strictly positive. This is realized by replacing program variables in path conditions by fresh *logic* variables and prepending to the symbolic store an update which describes the replacement. For $(\mathtt{x} > 0, \mathit{Skip})$, we obtain (in slightly unclean notation) "$\bigcup_{\beta} (v > 0, \mathtt{x} := v)$".

Definition 5.7 (Semantics of Symbolic Traces)**.** The trace valuation function *tval* is defined as in Fig. 5.1 for symbolic traces ϖ, ϖ', where $\mathit{fpv}(C)$ is the tuple of free program variables in path condition C, and $\overrightarrow{v}$ is a tuple of fresh logic variables of same length and types as $\mathit{fpv}(C)$. ◊

By this definition, the semantics of the empty symbolic trace $\bot$ is the empty concrete trace ε. This is different from the *empty trace set* to which an SES with unsatisfiable path condition evaluates: $\mathit{tval}(K, \sigma|(\text{false}, \mathscr{U})) = \emptyset$. The empty trace is the neutral element of trace concatenation ($\varepsilon\tau = \tau$ for all $\tau \in \mathit{Traces}$); however, concatenations with one element

$$\begin{aligned}
tval(K,\sigma|\bot) &:= \{\varepsilon\} \\
tval(K,\sigma|\top) &:= \{\tau \in Traces \mid finite(\tau)\} \\
tval(K,\sigma|\top^\infty) &:= Traces \\
tval(K,\sigma|(C,\mathscr{U})) &:= \bigcup_\beta \Big\{val\big(K,\sigma,\beta|\big(fpv(C) := \overrightarrow{v}\big) \circ \mathscr{U}\big)(\sigma) \mid \\
&\qquad K,\sigma,\beta \models \bigwedge\big(C[\overrightarrow{v}/fpv(C)]\big)\Big\} \\
tval(K,\sigma|\neg\varpi) &:= \overline{tval(K,\sigma|\varpi)} \\
tval\big(K,\sigma|\varpi;\varpi'\big) &:= \{\tau \in tval(K,\sigma|\varpi) \mid \neg finite(\tau)\} \cup \\
&\qquad \big\{\tau\tau' \mid \tau \in tval(K,\sigma|\varpi) \wedge finite(\tau) \wedge \tau' \in tval\big(K,\sigma|\varpi'\big)\big\} \\
tval\big(K,\sigma|\varpi+\varpi'\big) &:= tval(K,\sigma|\varpi) \cup tval\big(K,\sigma|\varpi'\big) \\
tval(K,\sigma|\varpi^*) &:= \{\tau_1\tau_2\cdots\tau_n \mid n \geq 0 \wedge \tau_i \in tval(K,\sigma|\varpi)\} \\
tval(K,\sigma|\varpi^\omega) &:= \{\tau_1\tau_2\cdots \mid \tau_i \in tval(K,\sigma|\varpi)\}
\end{aligned}$$

Figure 5.1: Trace Valuation Function *tval*

evaluating to the empty set of traces evaluate to the empty trace set, too. This is used for the choice operator: If path conditions are mutually exclusive, exactly one of $tval(K,\sigma|\varpi)$ and $tval\big(K,\sigma|\varpi'\big)$ is non-empty. The semantics of a PDL test $\varphi?$ can be realized by adding φ to the path condition of the subsequent SES.

We provide two examples for translations to symbolic traces: First, for first-order and LTL formulas, and second, for programs of a WHILE language. Given a TDL construct $\mathscr{C}$, we write $\mathscr{C}^s$ for its symbolic trace translation.

Example 5.6 (Symbolic Translation of Formulas)**.** A first-order formula φ with big-step semantics is translated to the symbolic trace $\varphi^s := \top;(\varphi)$ representing all finite traces ending in a state satisfying φ. For LTL formulas, we obtain the following translations:

$$(\varphi^{ltl})^s = (\varphi);\top^\infty \qquad \bigcirc\xi^s = (\text{true});\xi^s \qquad \Diamond\xi^s = \top;\xi^s;\top^\infty$$
$$\Box\xi^s = (\xi^{s\prime})^\omega \qquad \zeta\mathsf{U}\xi^s = (\zeta^s)^*;\xi^s;\top^\infty$$

where $\xi^{s\prime}$ is the result of removing trailing occurrences of $\top^\infty$. ◊

To translate programs, we define the *symbolic trace composition operator* $\bowtie$: *STraces* $\times$ *STraces* $\rightharpoonup$ *STraces* applying the *final* states of the first trace to *all* states of the second trace.

$$
\begin{aligned}
\bot \bowtie \varpi &:= \varpi \\
\top \bowtie \varpi &:= \varpi \\
\top^{\infty} \bowtie \varpi &:= \varpi \\[4pt]
(C, \mathscr{U}) \bowtie \bot &:= \bot \\
(C, \mathscr{U}) \bowtie \left(C', \mathscr{U}'\right) &:= \left(C \cup \{\mathscr{U}\}C', \mathscr{U} \circ \mathscr{U}'\right) \\
(C, \mathscr{U}) \bowtie (\varpi; \varpi') &:= ((C, \mathscr{U}) \bowtie \varpi); ((C, \mathscr{U}) \bowtie \varpi') \\
(C, \mathscr{U}) \bowtie (\varpi + \varpi') &:= ((C, \mathscr{U}) \bowtie \varpi) + ((C, \mathscr{U}) \bowtie \varpi') \\
(C, \mathscr{U}) \bowtie \varpi^{*} &:= ((C, \mathscr{U}) \bowtie \varpi)^{*} \\
(C, \mathscr{U}) \bowtie \varpi^{\omega} &:= ((C, \mathscr{U}) \bowtie \varpi)^{\omega} \\[4pt]
(\varpi; \varpi') \bowtie \varpi'' &:= \varpi' \bowtie \varpi'' \\
(\varpi + \varpi') \bowtie \varpi'' &:= (\varpi \bowtie \varpi'') + (\varpi' \bowtie \varpi'') \\[4pt]
\varpi^{*} \bowtie \varpi' &:= \varpi \bowtie \varpi' \\
\varpi^{\omega} \bowtie \varpi' &:= \varpi \bowtie \varpi'
\end{aligned}
$$

Figure 5.2: Symbolic Trace Composition Operator

For instance, the result of $(s_1;s_2) \bowtie (s_3;s_4)$ is $(s_2 \bowtie s_3);(s_2 \bowtie s_4)$. Individual symbolic states are merged by "applying" the first to the second state: $(C,\mathcal{U}) \bowtie \left(C',\mathcal{U}'\right)$ is equivalent to $\left(C \cup \{\mathcal{U}\}C', \mathcal{U} \circ \mathcal{U}'\right)$. The idea is similar to partial evaluation: A given symbolic trace is specialized according to some prefix. The complete inductive definition (over the structure of symbolic traces) is shown in Fig. 5.2. Note that this operator is a *partial* function; e.g., $(C,\mathcal{U}) \bowtie \top^\infty$ is not defined.

In the following, we use the shorthand $\varpi \diamond \varpi'$ for $\varpi;(\varpi \bowtie \varpi')$.

Example 5.7 (Symbolic Tranlation of WHILE Programs)**.** For a WHILE language consisting assignments "$\mathtt{x}$=e", sequential composition $p_1 \mathtt{;} p_2$, conditional $\mathbf{if}(g)\ p_1\ \mathbf{else}\ p_2$, and assertion $\mathbf{assert}(\varphi)$, we can define the symbolic translation as follows (we postpone the discussion of the more complicated loop statement):

$$\begin{aligned}
(\mathtt{x}{=}e)^s &= (\mathtt{x} := e)\\
(p_1\mathtt{;}p_2)^s &= {p_1}^s \diamond {p_2}^s\\
(\mathbf{if}(g)\ p_1\ \mathbf{else}\ p_2)^s &= (((g)\diamond(p_1)^s)) + (((\neg g)\diamond(p_2)^s))\\
(\mathbf{assert}(\varphi))^s &= (\varphi)
\end{aligned}$$

◊

The following concrete example shows how to translate a simple loop-free program to a symbolic trace. It also exemplifies the application of symbolic trace abstraction.

Example 5.8 (Symbolic Trace Translation and Abstraction for Programs)**.** Consider the following program p (in Java syntax) computing the difference of two integers `a` and `b`:

```
res=0; if (b < a) { tmp=a; a=b; b=tmp; } res=b-a;
```

The symbolic trace translation p^s has the shape

$$\begin{aligned}
(\mathtt{res} := 0) \diamond \big(((\mathtt{b} < \mathtt{a}) \diamond ((\mathtt{tmp} := \mathtt{a}) \diamond (\mathtt{a} := \mathtt{b}) \diamond (\mathtt{b} := \mathtt{tmp}))) +\\
((\mathtt{b} \geq \mathtt{a}) \diamond \bot)\big) \diamond (\mathtt{res} := \mathtt{b} - \mathtt{a})
\end{aligned}$$

By evaluating the composition operator $\diamond$ and finally simplifying the update concatenations, we obtain the final symbolic trace as shown in Fig. 5.3. We omitted an initial state "(true)" that would have to be prepended. Applying the symbolic equivalent of big-step abstraction α^{inf}_{big} to the resulting trace (with initial state) leads to

$$(\text{true});\top^\infty;((\mathtt{b} < \mathtt{a}, \mathtt{tmp} := \mathtt{a} \,||\, \mathtt{a} := \mathtt{b} \,||\, \mathtt{b} := \mathtt{a} \,||\, \mathtt{res} := \mathtt{a} - \mathtt{b}) + (\mathtt{b} \geq \mathtt{a}, \mathtt{res} := \mathtt{b} - \mathtt{a}))$$

$$
\begin{aligned}
&(\mathtt{res} := 0) \diamond \big(((\mathsf{b} < \mathsf{a}) \diamond ((\mathtt{tmp} := \mathsf{a}) \diamond (\mathsf{a} := \mathsf{b}) \diamond (\mathsf{b} := \mathtt{tmp}))) + \\
&\qquad ((\mathsf{b} \geq \mathsf{a}) \diamond \bot)\big) \diamond (\mathtt{res} := \mathsf{b} - \mathsf{a}) \\
&= (\mathtt{res} := 0) \diamond \big(((\mathsf{b} < \mathsf{a}) \diamond (\ (\mathtt{tmp} := \mathsf{a}); \\
&\qquad\qquad ((\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b})); \\
&\qquad\qquad ((\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b}) \circ (\mathsf{b} := \mathtt{tmp})))) + \\
&\qquad (\mathsf{b} \geq \mathsf{a})\big) \diamond (\mathtt{res} := \mathsf{b} - \mathsf{a}) \\
&= (\mathtt{res} := 0) \diamond \big(((\mathsf{b} < \mathsf{a}, (\mathtt{tmp} := \mathsf{a})); \\
&\qquad (\mathsf{b} < \mathsf{a}, (\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b})); \\
&\qquad (\mathsf{b} < \mathsf{a}, (\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b}) \circ (\mathsf{b} := \mathtt{tmp}))) + \\
&\qquad (\mathsf{b} \geq \mathsf{a})\big) \diamond (\mathtt{res} := \mathsf{b} - \mathsf{a}) \\
&= (\mathtt{res} := 0); \big(((\mathsf{b} < \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{tmp} := \mathsf{a})); \\
&\qquad (\mathsf{b} < \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b})); \\
&\qquad (\mathsf{b} < \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b}) \circ (\mathsf{b} := \mathtt{tmp}))) + \\
&\qquad (\mathsf{b} \geq \mathsf{a}, \mathtt{res} := 0)\big) \diamond (\mathtt{res} := \mathsf{b} - \mathsf{a}) \\
&= (\mathtt{res} := 0); ((\mathsf{b} < \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{tmp} := \mathsf{a})); \\
&\qquad (\mathsf{b} < \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b})); \\
&\qquad (\mathsf{b} < \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b}) \circ (\mathsf{b} := \mathtt{tmp})); \\
&\qquad (\mathsf{b} < \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{tmp} := \mathsf{a}) \circ (\mathsf{a} := \mathsf{b}) \circ \\
&\qquad\qquad (\mathsf{b} := \mathtt{tmp}) \circ (\mathtt{res} := \mathsf{b} - \mathsf{a})) + \\
&\qquad ((\mathsf{b} \geq \mathsf{a}, \mathtt{res} := 0); (\mathsf{b} \geq \mathsf{a}, (\mathtt{res} := 0) \circ (\mathtt{res} := \mathsf{b} - \mathsf{a}))) \\
&= (\mathtt{res} := 0); ((\mathsf{b} < \mathsf{a}, \mathtt{res} := 0 \,||\, \mathtt{tmp} := \mathsf{a}); \\
&\qquad (\mathsf{b} < \mathsf{a}, \mathtt{res} := 0 \,||\, \mathtt{tmp} := \mathsf{a} \,||\, \mathsf{a} := \mathsf{b}); \\
&\qquad (\mathsf{b} < \mathsf{a}, \mathtt{res} := 0 \,||\, \mathtt{tmp} := \mathsf{a} \,||\, \mathsf{a} := \mathsf{b} \,||\, \mathsf{b} := \mathsf{a}); \\
&\qquad (\mathsf{b} < \mathsf{a}, \mathtt{tmp} := \mathsf{a} \,||\, \mathsf{a} := \mathsf{b} \,||\, \mathsf{b} := \mathsf{a} \,||\, \mathtt{res} := \mathsf{a} - \mathsf{b})) + \\
&\qquad ((\mathsf{b} \geq \mathsf{a}, \mathtt{res} := 0); (\mathsf{b} \geq \mathsf{a}, \mathtt{res} := \mathsf{b} - \mathsf{a}))
\end{aligned}
$$

Figure 5.3: Derivation of Final Symbolic Trace in Example 5.8

Intuitively, this trace represents all traces of at least two states, where the last one is represented by the SESs in the choice expression. In particular, all of these satisfy the postcondition $\texttt{res} \geq 0$. The symbolic equivalent of α^{fin}_{big} would insert $\top$ instead of $\top^{\infty}$, if there are no $\top^{\infty}$ or $\circ^{\omega}$ elements in the abstracted trace. We can also symbolically represent, for example, the combined abstraction $\alpha^{inf}_{big} \circ \alpha_{\{\texttt{res}\}}$, which produces the trace $(\mathrm{true}); \top^{\infty}; ((\mathsf{b} < \mathsf{a}, \texttt{res} := \mathsf{a} - \mathsf{b}) + (\mathsf{b} \geq \mathsf{a}, \texttt{res} := \mathsf{b} - \mathsf{a}))$. ◊

To connect the worlds of MTL and STL, we define two properties of symbolic trace translations. For a construct $\mathscr{C}$, its translation $\mathscr{C}^s$ is *sound* if it represents fewer traces than $\mathscr{C}$. Conversely, it is *complete* if it represents more traces. These definitions allow to conclude the validity of $\mathscr{C}$ from the validity of $\vdash \neg\mathscr{C}^s$ for sound translations $\mathscr{C}^s$, and vice versa for complete translations. A translation is *precise* if it is both sound and complete.

Definition 5.8 (Sound, Complete and Precise Symbolic Trace Translations)**.** Let $\mathscr{C}$ be a TDL construct and $\mathscr{C}^s$ its translation to regular symbolic traces. The translation is *sound* iff for all K, σ it holds that $tval(K,\sigma|\mathscr{C}^s) \subseteq tval(K,\sigma|\mathscr{C})$. It is *complete* iff for all K, σ it holds that $tval(K,\sigma|\mathscr{C}^s) \supseteq tval(K,\sigma|\mathscr{C})$. We call translations which are *both* sound and complete *precise*. This extends to symbolic translations of *trace abstractions* as follows: Let $\mathscr{C}^s$ be a sound translation of $\mathscr{C}$. Then, $\mathfrak{a}$ is a sound translation of α if $\mathfrak{a}(\mathscr{C}^s) \subseteq \alpha(\mathscr{C})$, and similarly for completeness/precision. ◊

To formalize the intuition about sound and complete symbolic trace translations, we continue by also formally introducing syntax and semantics of STL sequents.

Definition 5.9 (STL Sequents)**.** An STL *sequent* has the form $\varpi_1,\ldots,\varpi_n \vdash \varpi_1,\ldots,\varpi_m$ for $\varpi_i, \varpi_j \in \mathit{STraces}$. Order and multiple occurrences of antecedents ϖ_i and succedents ϖ_j are irrelevant. The trace semantics of a sequent is defined by

$$tval(K,\sigma|\varpi_1,\ldots,\varpi_n \vdash \varpi_1,\ldots,\varpi_m) := \bigcup_{1=i}^{n} tval(K,\sigma|\varpi_i) \cup \overline{\bigcap_{1=j}^{m} tval\big(K,\sigma|\varpi_j\big)}$$

Instead of writing "$\models \Gamma \vdash \Delta$", we simply write $\Gamma \vdash \Delta$ and say that the sequent is *valid*. ◊

The notion of *validity* for a sequent $\Gamma \vdash \Delta$ already follows from the concept of validity of MTL: We call the sequent valid iff for all K, σ it holds that $tval(K,\sigma|\Gamma \vdash \Delta) = \mathit{Traces}$.

The following lemma establishes the connection of the validity of *STL sequents* and *MTL formulas* via the notions of soundness and completeness of symbolic trace translations.

Lemma 5.5. *Let $\mathscr{C}$ be TDL construct and $\mathscr{C}^s$ its symbolic translation. It holds that*

(1) If $\mathscr{C}^s$ is sound, then $\vdash \neg\mathscr{C}^s$ implies $|\models \mathscr{C}$.

(2) If $\mathscr{C}^s$ is complete, then $|\models \mathscr{C}$ implies $\vdash \neg\mathscr{C}^s$.

Proof. $\vdash \neg\mathscr{C}^s$ is equivalent to $tval(K,\sigma|\vdash\neg\mathscr{C}^s) = Traces$ and thus to $tval(K,\sigma|\mathscr{C}^s) = Traces$. For Item (1), we have to show that $tval(K,\sigma|\mathscr{C}^s) = Traces$ implies $tval(K,\sigma|\mathscr{C}) = Traces$, which is a consequence from $tval(K,\sigma|\mathscr{C}^s) \subseteq tval(K,\sigma|\mathscr{C})$ and the fact that *Traces* constitutes the whole universe of traces. The argument for Item (2) is symmetric. $\square$

As the semantics of a trace modality formula $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ is

$$\overline{\alpha\left(tval\left(K,\sigma|\mathscr{C}_{impl}\right)\right)} \cup \alpha\left(tval\left(K,\sigma|\mathscr{C}_{spec}\right)\right),$$

it is quite straightforward to come up with its canonical translation into symbolic traces, which has a similar shape:

$$\neg\mathfrak{a}\left({\mathscr{C}_{impl}}^s\right) + \mathfrak{a}\left({\mathscr{C}_{spec}}^s\right)$$

The following lemma describes sufficient conditions for this to be a sound and/or complete translation of a trace modality formula.

Lemma 5.6 (Symbolic Translation of the Trace Modality). *Let $\mathscr{C}_{impl}$, $\mathscr{C}_{spec}$ be two TDL constructs, and α be a trace abstraction. Then, $\neg\mathfrak{a}\left({\mathscr{C}_{impl}}^s\right) + \mathfrak{a}\left({\mathscr{C}_{spec}}^s\right)$ is a*

(1) sound *translation of $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ if $\mathfrak{a}$ is a* precise *translation of α, ${\mathscr{C}_{impl}}^s$ is a* complete *translation of $\mathscr{C}_{impl}$, and ${\mathscr{C}_{spec}}^s$ is a* sound *translation of $\mathscr{C}_{spec}$.*

(2) complete *translation of $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ if $\mathfrak{a}$ is a* precise *translation of α, ${\mathscr{C}_{impl}}^s$ is a* sound *translation of $\mathscr{C}_{impl}$, and ${\mathscr{C}_{spec}}^s$ is a* complete *translation of $\mathscr{C}_{spec}$.*

More generally, it is a

(3) sound *translation of $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$ if*

$$tval\left(K,\sigma|\mathfrak{a}\left({\mathscr{C}_{impl}}^s\right)\right) \supseteq \alpha\left(tval\left(K,\sigma|\mathscr{C}_{impl}\right)\right), \textit{ and}$$
$$tval\left(K,\sigma|\mathfrak{a}\left({\mathscr{C}_{spec}}^s\right)\right) \subseteq \alpha\left(tval\left(K,\sigma|\mathscr{C}_{spec}\right)\right).$$

(4) complete *translation of* $[\mathscr{C}_{impl} \Vdash_{\alpha} \mathscr{C}_{spec}]$ *if*

$$tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{impl}{}^{s}\big)\big) \subseteq \alpha\big(tval\big(K,\sigma|\mathscr{C}_{impl}\big)\big), \textit{ and}$$
$$tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{spec}{}^{s}\big)\big) \supseteq \alpha\big(tval\big(K,\sigma|\mathscr{C}_{spec}\big)\big). \qquad \Diamond$$

Proof. The semantics of the translation is

$$tval\big(K,\sigma|\neg\mathfrak{a}\big(\mathscr{C}_{impl}{}^{s}\big)+\mathfrak{a}\big(\mathscr{C}_{spec}{}^{s}\big)\big) = \overline{tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{impl}{}^{s}\big)\big)} \cup tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{spec}{}^{s}\big)\big).$$

The translation is *sound* iff it represents fewer traces, i.e.,

$$\overline{tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{impl}{}^{s}\big)\big)} \cup tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{spec}{}^{s}\big)\big) \subseteq \overline{\alpha\big(tval\big(K,\sigma|\mathscr{C}_{impl}\big)\big)} \cup \alpha\big(tval\big(K,\sigma|\mathscr{C}_{spec}\big)\big) \tag{5.2}$$

This is the case if it holds that

$$tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{impl}{}^{s}\big)\big) \supseteq \alpha\big(tval\big(K,\sigma|\mathscr{C}_{impl}\big)\big), \text{ and} \tag{5.3}$$
$$tval\big(K,\sigma|\mathfrak{a}\big(\mathscr{C}_{spec}{}^{s}\big)\big) \subseteq \alpha\big(tval\big(K,\sigma|\mathscr{C}_{spec}\big)\big) \tag{5.4}$$

which proves the more general sufficient Condition (3). For Condition (1), it holds that $tval\big(K,\sigma|\mathscr{C}_{impl}{}^{s}\big) \supseteq tval\big(K,\sigma|\mathscr{C}_{impl}\big)$ since $\mathscr{C}_{impl}{}^{s}$ is complete. Since furthermore, $\mathfrak{a}$ is precise, Eq. (5.3) follows. On the other hand, since $\mathscr{C}_{spec}{}^{s}$ is sound, it holds that $tval\big(K,\sigma|\mathscr{C}_{spec}{}^{s}\big) \subseteq tval\big(K,\sigma|\mathscr{C}_{spec}\big)$. Since furthermore, $\mathfrak{a}$ is precise, Eq. (5.4) follows. Observe that for Eqs. (5.3) and (5.4) to hold simultaneously, it is necessary that $\mathfrak{a}$ is precise; otherwise, it would have to be complete for Eq. (5.3) and sound for Eq. (5.4). However, Eqs. (5.3) and (5.4) are *sufficient*, but *not necessary* for Eq. (5.2) to hold, as there could be a complex interplay between the elements of the unions.

For *completeness*, the argument is symmetric. □

While the conditions expressed in Lem. 5.6 are only sufficient, but not necessary, it is hardly possible to name a general necessary condition which is *helpful*. For instance, Eq. (5.2) is a necessary and sufficient condition for soundness of the translation of the trace modality; however, we cannot easily derive conclusions from it for the design of symbolic translations of TDL constructs and trace abstractions.

Using Lem. 5.6 implies that symbolic translations of trace abstractions *have to* be precise. Luckily, it is usually possible to come up with precise translations, as Example 5.8 shows for the big-step and observation abstractions. For data abstraction, it is possible for suitably designed abstract domains. In general, we recommend to not use trace abstractions for which there does not exist a precise symbolic representation.

5.4.2 Symbolic Translation of Loops

There is another implication of Lem. 5.6, which is related to the symbolic translation of looping programs. To translate a loop statement $\mathtt{\textbf{while}}(b)\ p$, we can apply techniques known from Symbolic Execution: (Bounded) loop unwinding and invariant reasoning. The conceptually simplest solution is loop unwinding, which pulls out one iteration:

$$(\mathtt{\textbf{while}}(b)\ p)^s := (\mathtt{\textbf{if}}(b)\ \{\ \mathtt{p;}\ \mathtt{\textbf{while}}(b)\ p\ \})^s$$

This would, however, yield *infinite symbolic traces* which are not defined. The alternative approach is based on a summary of the loop's behavior called a *loop invariant*. Loop invariants (see also Sect. 2.3) *overapproximate* the effects of the loop and are always maintained by complete iterations: If a formula *Inv* holds directly before first entering the loop, and before each further iteration, *Inv* is called a loop invariant. We may then "replace" the loop by *Inv* and continue symbolically executing the remaining program. Here, we simply *assume* that *Inv* indeed is a loop invariant of the **while** statement. Then, we can translate the statement as follows to a symbolic trace:

$$(\mathtt{\textbf{while}}(b)\ p)^s := (\{\mathcal{U}_{havoc}\}(Inv \land b), \mathcal{U}_{havoc})^\omega; (\{\{\mathcal{U}_{havoc}\}(Inv \land \neg b)\}, \mathcal{U}_{havoc}) \tag{5.5}$$

where the update $\mathcal{U}_{havoc}$ anonymizes all locations which are written in p. The meaning of this symbolic trace is "either, we continue forever with an infinite number of loop iterations all satisfying the invariant formula *Inv*, or after a finite number of such iterations, we end in some state where the loop guard evaluates to **false** and the invariant still holds". In that final SES, we either have to "wipe" the whole store, or at least the part that might be written by the loop body, since their values will generally not be maintained. It is the job of the loop invariant to encode with sufficient precision their values in the post state. Additionally to the invariant, we can specify a *variant term* which proves termination of the loop. Then, we can replace the $(\cdots)^\omega$ by $(\cdots)^*$.

This understanding of loop invariants as an *abstraction* of the loop's behavior implies that Eq. (5.5) is a *complete* translation of $\mathtt{\textbf{while}}(b)\ p$. According to Lem. 5.6, this is convenient when using loop invariant-based symbolic execution of loops for programs in the *implementation* part of the trace modality—everything else would also be unexpected, considering that using loop invariants is a common approach for abstracting implementations that should be checked against a specification. However, in relational verification scenarios where the validity of an expression $[p_{impl} \Vdash_\alpha p_{spec}]$ should be shown, loop invariants used in the *specification program* p_{spec} have to be *sound*, otherwise the whole translation of the trace modality formula will not be sound. Therefore, we can either apply a technique like *bounded* unwinding up to a fixed number of loop iterations, which is likely to affect *completeness*, i.e., will render problems unprovable. Alternatively, we can use *underapproximating* loop invariants for specification programs; then, we also have to use big-step abstraction, since loop invariants in general abstract from concrete traces. This is a use case for an application of Conditions (3) and (4) of Lem. 5.6 instead of the more specific Conditions (1) and (2). Using loop invariants for symbolic trace translations can never be sound (in the sense of Def. 5.8); however, symbolic abstractions of these translations can be sound w.r.t. abstractions of the translated constructs.

Of course, fully *precise* loop invariants are acceptable, too. In fact, program *equivalence* proofs (which involve showing both directions of the trace modality) strictly require precise invariants, or, in other words, "the strongest possible functional loop invariant" [BU18]. This is because each program has to use an overapproximating invariant for one direction and an underapproximating one for the other, and only a precise invariant can satisfy both classifications. One can avoid the necessity of strongest functional invariants by using *coupling invariants*. Then, a different translation of the trace modality, e.g., based on product program constructions [BCK11], would be needed (and could be constructed).

Example 5.9 (Underapproximating Loop Invariants). To motivate the use of strictly underapproximating loop invariants, i.e., loop descriptions of a *strict subset* of possible behavior, assume a WHILE language with a choice statement "p_1 **`or`** p_2", which nondeterministically chooses either p_1 or p_2 and then executes it. Let

$$p_{impl} \coloneqq \texttt{\textbf{while} (i>0) i--;}$$
$$p_{spec} \coloneqq \texttt{\textbf{while} (i>0) \{ i--; \textbf{or} \{i--; i--\}; \}}$$

The traces for p_{impl} are

$$\bigcup_{K,\sigma} \big(tval\big(K,\sigma|p_{impl}\big)\big) = \{\sigma\sigma \mid \sigma(\mathtt{i}) \le 0\} \cup \{\sigma_1\sigma_2\cdots\sigma_n \mid \sigma = \sigma_1 \land \sigma(\mathtt{i}) > 0 \land \sigma_n(\mathtt{i}) = 0 \land \forall 0 < k < n; \sigma_k(\mathtt{i}) - \sigma_{k+1}(\mathtt{i}) = 1\}\,.$$

All of these traces are finite; either, the loop is not entered because `i` is initially zero or negative, or the value of `i` is decreased by 1 in every iteration, until it equals 0 and the program terminates. The traces of p_{spec} have the following shape:

$$\bigcup_{K,\sigma}\big(tval\big(K,\sigma|p_{spec}\big)\big) = \{\sigma\sigma \mid \sigma(\mathtt{i}) \leq 0\} \cup \\ \{\sigma_1\sigma_2\cdots\sigma_n \mid \sigma = \sigma_1 \wedge \sigma(\mathtt{i}) > 0 \wedge (\sigma_n(\mathtt{i}) = 0 \vee \sigma_n(\mathtt{i}) = -1) \wedge \\ \forall 0 < k < n; 1 \leq \sigma_k(\mathtt{i}) - \sigma_{k+1}(\mathtt{i}) \leq 2\}\,.$$

This is a superset of the traces of p_{impl}: If in every iteration of p_{spec} the left branch of the **`or`** is chosen, we obtain a trace of p_{impl} for every initial state σ_0. It is a *strict* superset since it also contains traces which terminate quicker or have final states where `x` attains the value -1 instead of 0.

A *precise* loop invariant for p_{impl} is $\mathtt{i} \geq 0$; together with the negated guard, $\mathtt{i} \leq 0$, this allows to deduce the *exact* final value of `i`, 0. The simplest strictly overapproximating (and therefore imprecise) loop invariant is "true"; another, more precise example is $\mathtt{i} \geq -1$. The latter is in fact a precise invariant for p_{spec}, which we can use to abstract the loop if we also use big-step abstraction. However, in a proof of $[p_{impl} \Vdash_{\alpha^{fin}_{big}} p_{spec}]$, we might also use $\mathtt{i} \geq 0$ as invariant for p_{spec}. This invariant describes *some* of the traces of p_{spec}, but not all—it is an *underapproximating* loop invariant, which is, however, "abstract enough", since it describes all traces of p_{impl}. It would be unsound to use it in a proof of the converse $[p_{spec} \Vdash_{\alpha^{fin}_{big}} p_{impl}]$, which we then could prove *although it is invalid*. $\Diamond$

We defined the syntax and semantics of STL, specifically of regular symbolic traces and sequents. Furthermore, Lems. 5.5 and 5.6 connect MTL and STL, allowing to phrase validity of MTL formulas as validity of STL sequents under certain conditions on the translation to symbolic traces. In the next section, we define the STL calculus by which we can mechanically prove the validity of STL sequents.

5.4.3 Rules and Proofs

The calculus of STL for proving judgments $\varpi_1,\ldots,\varpi_n \vdash \varpi_1,\ldots,\varpi_m$ for symbolic traces $\varpi_1,\ldots,\varpi_n$ and $\varpi,\ldots,\varpi_m$ is a *sequent calculus*. As usual, rules have the form

$$\textsf{ruleName}\ \frac{P_1 \quad \ldots \quad P_k}{Concl}$$

where the sequents $P_1, \dots, P_k$, for $k \geq 0$, are the premises of the rule "ruleName" and the sequent *Concl* is its conclusion. Rules contain the schematic variables Γ, Δ for sets of symbolic traces, $\varpi, \varpi', \dots$ for individual symbolic traces, C, C' for sets of formulas, and $\mathcal{U}, \mathcal{U}', \dots$ for updates. An instance of a rule is obtained by consistently instantiating schematic variables in the premise and conclusion. Rule applications are "bottom-up": Starting from a root sequent, we look for rules which can be instantiated such that their conclusion matches the sequent; we are then left with the task to prove the instantiations of the premises of the rule. The following definition stipulates some standard terms of sequent calculi proofs (for STL sequents).

Definition 5.10 (STL Proof Tree)**.** An STL *proof tree* is a tree starting from a root node shown at the bottom. Each node is labeled with an STL sequent or the symbol $*$. If an inner node is labeled with a sequent, there is an instance of the rule which can be instantiated s.t. its conclusion matches the sequent, and its premises match the children of the node. A branch in the proof tree is called *closed* if its leaf is labeled with $*$. A proof tree with only closed branches is also called closed. We say that an STL sequent $\Gamma \vdash \Delta$ can be *derived* if there is a closed proof tree whose root is labeled with $\Gamma \vdash \Delta$. ◊

Figure 5.4 shows the rules of the STL sequent calculus. Note that we omitted elimination rules for the empty trace $\bot$; instead, we assume that any (sub)expression $\varpi;\bot$ and $\bot;\varpi$ is simplified to ϖ, as $\bot$ is the neutral element of sequential symbolic trace composition. Four rules, +left, +right, ¬left and ¬right, perform simplifications based on the set-theoretic foundation of the semantics of STL sequents: Since the semantics of the antecedent is a disjunction of sets, we can replace the choice operator "+" by a comma in the antecedent (+left). In the succedent, on the other hand, we have to split the proof: A disjunction is a subset of another set iff *both* elements of the disjunction are subsets (+right). Symbolic complements can be resolved by shifting the trace to the other side of the sequent and removing the complement sign (¬left and ¬right).

Rule elim$\top^\infty$ instantiates an occurrence of $\top^\infty$ to any symbolic trace; the according rule elim$\top$ for $\top$ has a side condition preventing instantiations to potentially nonterminating traces, i.e., $\top^\infty$ as well as $(\varpi'')^\omega$, for any ϖ'', must not occur.

One rule eliminates a sequential composition simultaneously in the antecedent and succedent (elim;): If a prefix of an antecedent trace is more general than a prefix in the succedent trace, continue with the trailing traces. Note that we have to discard the context Γ, Δ in the premises, since the contrary would be unsound. Consequently, the rule is *incomplete*: It can turn out that by applying it, a previously valid problem is invalid afterward. Since there is no other rule for sequential composition, the whole calculus is

$$\text{+left}\ \frac{\Gamma, \varpi, \varpi' \ \vdash \ \Delta}{\Gamma, \varpi + \varpi' \ \vdash \ \Delta} \qquad \text{+right}\ \frac{\Gamma \ \vdash \ \varpi, \Delta \qquad \Gamma \ \vdash \ \varpi', \Delta}{\Gamma \ \vdash \ \varpi + \varpi' \Delta}$$

$$\neg\text{left}\ \frac{\Gamma \ \vdash \ \varpi, \Delta}{\Gamma, \neg\varpi \ \vdash \ \Delta} \qquad \neg\text{right}\ \frac{\Gamma, \varpi \ \vdash \ \Delta}{\Gamma \ \vdash \ \neg\varpi, \Delta}$$

$$\text{elim}\top^\infty\ \frac{\Gamma, (\top^\infty; \varpi), (\varpi'; \varpi) \ \vdash \ \Delta}{\Gamma, (\top^\infty; \varpi) \ \vdash \ \Delta}\ \varpi' \in \mathit{STraces} \qquad \text{elim}\top\ \frac{\Gamma, (\top; \varpi), (\varpi'; \varpi) \ \vdash \ \Delta}{\Gamma, (\top; \varpi) \ \vdash \ \Delta}\ \varpi' \text{ does not contain } (\varpi'')^\omega, \top^\infty$$

$$\text{elim;}\ \frac{\varpi \ \vdash \ \varpi' \qquad \varpi'' \ \vdash \ \varpi'''}{\Gamma, (\varpi; \varpi'') \ \vdash \ (\varpi'; \varpi'''), \Delta}\ (!)$$

$$\text{elim}^*{}_1\ \frac{\Gamma, (\varpi'; \varpi^*; \varpi'), (\varpi'; \varpi'') \ \vdash \ \Delta}{\Gamma, (\varpi'; \varpi^*; \varpi'') \ \vdash \ \Delta} \qquad \text{pull}^*\ \frac{\Gamma, (\varpi^*; \varpi'), (\varpi; \varpi^*; \varpi') \ \vdash \ \Delta}{\Gamma, (\varpi^*; \varpi') \ \vdash \ \Delta}$$

$$\text{elim}^*{}_2\ \frac{\varpi \ \vdash \ \varpi' \qquad \varpi'' \ \vdash \ \varpi'''}{\Gamma, (\varpi^*; \varpi'') \ \vdash \ ((\varpi')^*; \varpi'''), \Delta}\ (!)$$

$$\text{dupl}^*\ \frac{\Gamma, (\varpi^*; \varpi^*; \varpi') \ \vdash \ \Delta}{\Gamma, (\varpi^*; \varpi') \ \vdash \ \Delta} \qquad \text{dupl}^\omega\ \frac{\Gamma, (\varpi^\omega; \varpi^\omega; \varpi') \ \vdash \ \Delta}{\Gamma, (\varpi^\omega; \varpi') \ \vdash \ \Delta}$$

$$\text{elim}^\omega{}_1\ \frac{\Gamma, (\varpi'; \varpi^\omega; \varpi''), (\varpi'; \varpi'') \ \vdash \ \Delta}{\Gamma, (\varpi'; \varpi^\omega; \varpi'') \ \vdash \ \Delta} \qquad \text{pull}^\omega\ \frac{\Gamma, (\varpi^\omega; \varpi'), (\varpi; \varpi^\omega; \varpi') \ \vdash \ \Delta}{\Gamma, (\varpi^\omega; \varpi') \ \vdash \ \Delta}$$

$$\text{elim}^\omega{}_2\ \frac{\varpi \ \vdash \ \varpi' \qquad \varpi'' \ \vdash \ \varpi'''}{\Gamma, (\varpi^\omega; \varpi'') \ \vdash \ ((\varpi')^\omega; \varpi'''), \Delta}\ (!) \qquad \text{elim}^\omega{}_3\ \frac{\varpi \ \vdash \ \varpi' \qquad \varpi'' \ \vdash \ \varpi'''}{\Gamma, (\varpi^\omega; \varpi'') \ \vdash \ ((\varpi')^*; \varpi'''), \Delta}\ (!)$$

$$\text{cut}\ \frac{\Gamma, \varpi \ \vdash \ \Delta \qquad \Gamma, \neg\varpi \ \vdash \ \Delta}{\Gamma \ \vdash \ \Delta}$$

$$\text{closeSubsume}\ \frac{*}{\Gamma, (C, \mathcal{U}) \ \vdash \ \left(C', \mathcal{U}'\right), \Delta}\ (C, \mathcal{U}) \rhd (C', \mathcal{U}')$$

$$\text{close}\ \frac{*}{\Gamma, \varpi \vdash \varpi, \Delta} \qquad \text{close}\top^\infty\ \frac{*}{\Gamma, \top^\infty \vdash \Delta} \qquad \text{closeUnsat}\ \frac{*}{\Gamma \vdash (C, \mathcal{U}); \varpi, \Delta}\ (\text{false}, \mathit{Skip}) \rhd (C, \mathcal{U})$$

Figure 5.4: Calculus Rules for STL. Rules marked with (!) are incomplete.

incomplete. This reflects the complexity of checking regular expressions for subsumption. We discuss this issue at the end of this section.

Four rules address finite repetition ($\mathsf{elim}^*{}_1$, pull^*, dupl^* and $\mathsf{elim}^*{}_2$), four (potentially) *infinite* repetition ($\mathsf{elim}^\omega{}_1$, pull^ω, dupl^ω and $\mathsf{elim}^\omega{}_2$), and one ($\mathsf{elim}^\omega{}_3$) the combination of an infinite repetition in the antecedent and a finite one in the succedent. We can remove a repetition in the antecedent since zero repetitions are always possible ($\mathsf{elim}^*{}_1$ and $\mathsf{elim}^\omega{}_1$); furthermore, one can *unwind* one iteration (pull^* and pull^ω). We can duplicate a repetition ϖ^* to $\varpi^*; \varpi^*$ since both express an arbitrary, finite number of repetitions (dupl^*, or dupl^ω for infinite repetitions). To remove a repetition in the succedent, there has to be a more general one in the antecedent. This is addressed by $\mathsf{elim}^*{}_2$, $\mathsf{elim}^\omega{}_2$ and $\mathsf{elim}^\omega{}_3$, which are incomplete. Using dupl^*, dupl^ω can make sense before applying a rule like $\mathsf{elim}^*{}_2$, which *removes* the repetition (that could still be needed afterward).

Rule cut allows to proceed by *case distinction* at any point: In one branch, we augment the succedent by a symbolic trace ϖ, and in another by its complement $\neg\varpi$.

There are four closing rules: One for the situation where the same trace occurs in the antecedent and succedent (close), one for closing a branch with the most general trace expression $\top^\infty$ in the antecedent ($\mathsf{close}\top^\infty$), and a corresponding one for an unsatisfiable leading state in the succedent ($\mathsf{closeUnsat}$). The most interesting closing rule is $\mathsf{closeSubsume}$: Close a branch if a singleton trace in the antecedent is more general than, i.e., *represents more concrete states than*, a singleton trace in the succedent. Both $\mathsf{closeUnsat}$ and $\mathsf{closeSubsume}$ call an external *subsumption checker* for SESs: The judgment $s \rhd s'$ expresses that s represents more concrete states, or *subsumes*, s'.

5.4.4 Symbolic Subsumption

We first define a weak version of symbolic state subsumption, in which all uninterpreted symbols are treated universally by constructing the union over all structures and states.

Definition 5.11 (Weak Symbolic State Subsumption). Let $(C_1, \mathcal{U}_1)$, $(C_2, \mathcal{U}_2) \in \mathbb{S}_{SE}$. We say that $(C_1, \mathcal{U}_1)$ *weakly subsumes* $(C_2, \mathcal{U}_2)$ and write $(C_1, \mathcal{U}_1) \blacktriangleright (C_2, \mathcal{U}_2)$ iff

$$\bigcup_{K,\sigma} \left\{ val(K, \sigma | \mathcal{U}_1)(\sigma) \mid K, \sigma \models \bigwedge C_1 \right\} \supseteq \bigcup_{K,\sigma} \left\{ val(K, \sigma | \mathcal{U}_2)(\sigma) \mid K, \sigma \models \bigwedge C_2 \right\}$$

A *weak symbolic state subsumption checker* defines a relation $\blacktriangleright^? \in \mathbb{S}_{SE} \times \mathbb{S}_{SE}$ s.t.

$$(C_1, \mathcal{U}_1) \blacktriangleright^? (C_2, \mathcal{U}_2) \text{ implies } (C_1, \mathcal{U}_1) \blacktriangleright (C_2, \mathcal{U}_2) \qquad \lozenge$$

Using weak subsumption, it holds that $(c > 0, \mathtt{x} := c) \blacktriangleright (\emptyset, \mathtt{x} := 1)$, and even

$$(c > 0, \mathtt{x} := c) \blacktriangleright (c < 0, \mathtt{x} := -c).$$

Each SES is semantically encapsulated, rigid symbols are not “connected”. This notion is interesting, but too liberal for the STL calculus, since in the semantics of symbolic traces, the whole trace is interpreted by the *same* structure and state. Note that furthermore, in a system based on weak symbolic state subsumption, one cannot express a property like “throughout the execution trace, the program variable x attains some fixed value”, since that value would have to be *interpreted* (i.e., concrete) to be fixed in separated SESs.

We define a strong version in which rigid symbols are interpreted the same way in both symbolic states that are checked. The idea behind this strong symbolic state subsumption inspired the semantics of symbolic states as given in Def. 5.7: Everything is determined by the structure K and state σ *but the values of program variables in the path condition*.

Definition 5.12 (Strong Symbolic State Subsumption)**.** Let $(C_1, \mathcal{U}_1)$, $(C_2, \mathcal{U}_2) \in \mathbb{S}_{SE}$. For each $(C_i, \mathcal{U}_i)$, we define the state $\left(C_i', \mathcal{U}_i'\right)$ s.t.

$$\left(C_i', \mathcal{U}_i'\right) := \left(C_i[\overrightarrow{v}/\mathit{fpv}(C_i)], \left(\mathit{fpv}(C_i) := \overrightarrow{v}\right) \circ \mathcal{U}_i\right)$$

where $\mathit{fpv}(C_i)$ is the tuple of free program variables in path condition C_i and $\overrightarrow{v}$ a tuple of fresh logic variables of same length and types as $\mathit{fpv}(C_i)$. Observe that C_i' does not contain program variables. Then, we say that $(C_1, \mathcal{U}_1)$ *(strongly) subsumes* $(C_2, \mathcal{U}_2)$ and write $(C_1, \mathcal{U}_1) \rhd (C_2, \mathcal{U}_2)$ iff *for all structures K and states σ*, it holds that

$$\bigcup_{\beta} \left\{ val\left(K, \sigma, \beta | \mathcal{U}_1'\right)(\sigma) \mid K, \sigma, \beta \models \bigwedge C_1' \right\} \supseteq \bigcup_{\beta} \left\{ val\left(K, \sigma, \beta | \mathcal{U}_2'\right)(\sigma) \mid K, \sigma, \beta \models \bigwedge C_2' \right\}$$

A *(strong) symbolic state subsumption checker* defines a relation $\rhd^{?} \in \mathbb{S}_{SE} \times \mathbb{S}_{SE}$ s.t.

$$(C_1, \mathcal{U}_1) \rhd^{?} (C_2, \mathcal{U}_2) \text{ implies } (C_1, \mathcal{U}_1) \rhd (C_2, \mathcal{U}_2) \qquad \Diamond$$

Lemma 5.7 (Strong Implies Weak Subsumption)**.** *Let $(C_1, \mathcal{U}_1)$, $(C_2, \mathcal{U}_2) \in \mathbb{S}_{SE}$ be two symbolic states. Then, it holds that $(C_1, \mathcal{U}_1) \rhd (C_2, \mathcal{U}_2)$ implies $(C_1, \mathcal{U}_1) \blacktriangleright (C_2, \mathcal{U}_2)$.*

Proof. Let

$$\sigma' \in \bigcup_{K,\sigma} \left\{ val(K,\sigma|\mathcal{U}_2)(\sigma) \mid K,\sigma \models \bigwedge C_2 \right\} \tag{†}$$

We have to show that also

$$\sigma' \in \bigcup_{K,\sigma''} \left\{ val(K,\sigma|\mathcal{U}_1)(\sigma'') \mid K,\sigma'' \bigwedge C_1 \right\} \tag{⋆}$$

based on the assumption that $(C_1,\mathcal{U}_1) \rhd (C_2,\mathcal{U}_2)$. Let K,σ be the structure and state which gave rise to σ' in (†). Then, it has to hold that

$$\sigma' \in \bigcup_{\beta} \left\{ val\big(K,\sigma,\beta|\mathcal{U}_2'\big)(\sigma) \mid K,\sigma,\beta \models \bigwedge C_2' \right\}$$

since we can choose β such that it assigns values to free logic variables in C_2' according to the value of program variables in state σ. Due to $(C_1,\mathcal{U}_1) \rhd (C_2,\mathcal{U}_2)$, we obtain

$$\sigma' \in \bigcup_{\beta} \left\{ val\big(K,\sigma,\beta|\mathcal{U}_1'\big)(\sigma) \mid K,\sigma,\beta \models \bigwedge C_1' \right\}.$$

From this we can conclude (⋆) since we can, this time in reverse direction, choose σ'' such that it interprets free logic variables in C_1 according to β and apart from that equals σ; we can choose the same structure K.

A counterexample for the reverse direction is $(C_1,\mathcal{U}_1) := (c > 0, \mathtt{x} := c)$ and $(C_2,\mathcal{U}_2) := (\emptyset, \mathtt{x} := 1)$, for which it holds that $(C_1,\mathcal{U}_1) \blacktriangleright (C_2,\mathcal{U}_2)$, but not $(C_1,\mathcal{U}_1) \rhd (C_2,\mathcal{U}_2)$: There is a structure interpreting c s.t. it attains a value different from 1, i.e., $tval(K,\sigma|(C_1,\mathcal{U}_1))$ is the empty set or a singleton set where $\mathtt{x}$ attains a positive value different from 1, but $tval(K,\sigma|(C_2,\mathcal{U}_2))$ is a singleton set where $\mathtt{x}$ attains the value 1. For $(C_3,\mathcal{U}_3) := (\mathtt{x} > 0, Skip)$, we can show both $(C_3,\mathcal{U}_3) \blacktriangleright (C_2,\mathcal{U}_2)$ *and* $(C_3,\mathcal{U}_3) \rhd (C_2,\mathcal{U}_2)$. □

Subsumption checking is straightforward to define for common special cases. Consider the case of two SESs with empty stores: Let $(C_1, Skip) \rhd (C_2, Skip)$ be true if we can prove in a first-order solver that $C_2 \rightarrow C_1$. The case where only the symbolic store of the more general state is empty corresponds to standard postcondition verification: $(C_1, Skip) \rhd (C_2,\mathcal{U})$ holds if we can prove $C_2 \rightarrow \{\mathcal{U}\}C_1$, here in JavaDL syntax.

The matter gets more complicated in the general case of *two nonempty stores*. Reference [APV06] addresses the problem of subsumption checking (in an *underapproximation* scenario) for symbolic states of a symbolic heap and a set of constrains describing the valuations of primitively typed nodes. The authors propose a two-fold subsumption checking technique: First, they match heap configurations by a graph traversal algorithm (at this step, primitively typed nodes are ignored); next, they assert that primitive values and path conditions can be matched for suitable instantiations of symbolic names. A similar approach is pursued in [Xie+05]. It is not in the scope of this thesis to generally solve the—still underresearched—problem of symbolic state subsumption checking, in particular for complicated heap structures.

Instead, we propose a subsumption checking strategy which uses the JavaDL sequent calculus in the background. As explained in Sect. 2.2, we assume that there is a designated program variable `heap` of type *Heap*. We do not specially treat this variable, but instead defer the work of heap subsumption checking to the JavaDL heap theory.

The idea is to show that the path condition of the subsumed state, together with a system of equations describing the symbolic store, implies the path condition and equation system of the subsuming state. There is a weak and a strong variant: In the weak variant, we may instantiate abstract symbols arbitrarily. In the strong variant, we only replace free program variables in the path condition by logic variables, which are then existentially quantified over. Also in the weak variant, there may not appear program variables in the path condition. This is no serious restriction: For instance, an SES $(\mathsf{x} > 0, \mathcal{U})$ is equivalent (w.r.t. weak subsumption) to an SES $(c > 0, (\mathsf{x} := c) \circ \mathcal{U})$, for a fresh constant c.

Definition 5.13 (Weak JavaDL-Backed Subsumption Checker)**.** Let $s_1 = (C_1, \mathcal{U}_1)$, $s_2 = (C_2, \mathcal{U}_2)$ be two SESs, where C_1, C_2 do not contain program variables. We define the relation $\blacktriangleright^? \in \mathbb{S}_{SE} \times \mathbb{S}_{SE}$ s.t. $s_1 \blacktriangleright^? s_2$ iff there is a substitution θ of closed terms for uninterpreted constant symbols and updates for abstract update symbols (with compatible left- and right-hand sides) for which we can derive in the JavaDL calculus that $\vdash \left(C_2 \wedge t^{=}_{\mathcal{U}_2}\right) \rightarrow \theta\left(C_1 \wedge t^{=}_{\mathcal{U}_1}\right)$, where $t^{=}_{\mathcal{U}_i}$ is a clash-free translation of $\mathcal{U}_i$ into a conjunction of equations, s.t. $K, \sigma \models t^{=}_{\mathcal{U}_i}$ iff there is a $\sigma' \in \mathcal{S}$ s.t. $\sigma = val\big(K, \sigma' | \mathcal{U}_i\big)(\sigma')$. $\lozenge$

For example, $\mathsf{x} := 17 \| \mathsf{y} := c$ is translated to $\mathsf{x} \doteq 17 \wedge \mathsf{y} \doteq c$. Translating symbolic stores with abstract updates is only feasible if we assume that abstract update symbols only have concrete left-hand sides declared as "has-to" (cf. Sect. 4.2), or that $\mathcal{U}_2$ does not have abstract updates; those in $\mathcal{U}_1$ can be substituted by concrete ones. An abstract update $\mathcal{U}_{\mathsf{P}}(\mathsf{x}^!, \mathsf{y}^! :\approx \mathit{footprint})$ can be translated to $\mathsf{x} \doteq f_1^{\mathsf{P}}(\mathit{footprint}) \wedge \mathsf{y} \doteq f_2^{\mathsf{P}}(\mathit{footprint})$.[6] Conflicts in updates have to be first resolved by the "last update wins" semantics.

We forbid program variables in path conditions as otherwise, they could clash with the symbolic stores. Consider the SES $(\mathsf{x} > 0, \mathsf{x} := -1)$, which represents all concrete states

[6] We assume that structures K interpret symbols f_i^{P} according to their interpretation of $\mathcal{U}_{\mathsf{P}}$.

where $\mathtt{x}$ equals -1. The assignment in the store overrides the constraint in the path condition. However, after translating the store into an equation, we would obtain $\mathtt{x} > 0 \wedge \mathtt{x} \doteq -1$, which is unsatisfiable and therefore has a different meaning. First transforming the SES to $(\mathtt{c} > 0, (\mathtt{x} := c) \circ (\mathtt{x} := -1))$, which is equivalent to $(\mathtt{c} > 0, (\mathtt{x} := c) \,||\, (\{\mathtt{x} := c\}(\mathtt{x} := -1)))$ and therefore to $(\mathtt{c} > 0, \mathtt{x} := -1)$, leads to the correct result: $\mathtt{c} > 0 \wedge \mathtt{x} \doteq -1$. Since we can instantiate c arbitrarily, this formula is satisfied by all states in which $\mathtt{x}$ is -1. Observe that, in the weak setting, *path condition constraints are only effective if involved symbols occur in expressions of the symbolic store*.

Example 5.10 (Weak Symbolic State Subsumption Checking)**.** We first examine the properties of the weak JavaDL-based subsumption checker for the special cases of two and one empty stores mentioned above. For two SESs with empty stores, the translation of the stores to equations is trivial and results in "true"; we obtain $C_2 \to \theta\,(C_1)$. The application of θ only to C_1 (and not also to C_2) allows us to derive that, for example, the states $(c > d, \mathit{Skip})$ and $(d > c, \mathit{Skip})$ are in a subsumption relation, although $c > d \to d > c$ clearly does not hold. This is in accordance to Def. 5.11: It suffices to come up with *any* satisfying interpretation of C, which is considered when interpreting $\mathcal{U}$.

A postcondition $\mathtt{x} > 0$ is first transformed into an SES $(c > 0, \mathtt{x} := c)$. Given, for instance, an SES $(\emptyset, \mathtt{x} := 1)$, we have to show $\big(\mathrm{true} \wedge \mathtt{x} \doteq 1\big) \to \theta\,\big(c > 0 \wedge \mathtt{x} \doteq c\big)$. Instantiating the constant c to the variable $\mathtt{x}$ leads to the expected formula $\mathtt{x} \doteq 1 \to \mathtt{x} > 0$.

For a relational example, consider the problem

$$[\mathtt{x=17;y=-20;} \Vdash_{\alpha^{fin}_{big}} \mathtt{P}(\mathit{frameP} :\approx \mathit{footprintP})\,\mathtt{;x=17;}]$$

for an Abstract Statement $\mathtt{P}$. In an STL proof of this problem, we finally have to show

$$(\emptyset, \mathcal{U}_{\mathtt{P}}(\mathit{frameP} :\approx \mathit{footprintP}) \,||\, \mathtt{x} := 17) \blacktriangleright (\emptyset, \mathtt{x} := 17 \,||\, \mathtt{y} := -20)\,.$$

If there is no AE constraint excluding $\mathring{\mathtt{y}}$ from *frameP*, we can choose θ such that it substitutes the abstract update by $\mathtt{y} := -20$. Assignments to $\mathtt{x}$ by $\mathtt{P}$ are irrelevant since they are overwritten. $\Diamond$

The relation $\blacktriangleright^?$ defined in Def. 5.13 is a weak subsumption checker, i.e., whenever two SESs are contained in the relation, the first weakly subsumes the second. We prove the following lemma:

Lemma 5.8 (The Weak JavaDL-Backed Subsumption Checker is Correct)**.** *The relation* $\blacktriangleright^? \subseteq \mathbb{S}_{SE} \times \mathbb{S}_{SE}$ *defined in Def. 5.13 satisfies Def. 5.11.*

Proof. We have to show that

$$(C_1, \mathcal{U}_1) \blacktriangleright^? (C_2, \mathcal{U}_2) \text{ implies } (C_1, \mathcal{U}_1) \blacktriangleright (C_2, \mathcal{U}_2).$$

Assume that there is a derivation in JavaDL of $\vdash \left(\bigwedge C_2 \wedge t^{=}_{\mathcal{U}_2}\right) \rightarrow \theta\left(\bigwedge C_1 \wedge t^{=}_{\mathcal{U}_1}\right)$ for some substitution θ. Since the JavaDL calculus is sound, the entailment $\left(\bigwedge C_2 \wedge t^{=}_{\mathcal{U}_2}\right) \models \theta\left(\bigwedge C_1 \wedge t^{=}_{\mathcal{U}_1}\right)$ holds. If $\bigcup_{K,\sigma} \{val(K,\sigma|\mathcal{U}_2)(\sigma) \mid K,\sigma \models \bigwedge C_2\} = \emptyset$, we are done; therefore, assume that there is a state $\sigma \in \mathcal{S}$, a structure K and another state σ' s.t. $K,\sigma' \models \bigwedge C_2$ and $\sigma = val\big(K,\sigma'|\mathcal{U}_2\big)(\sigma')$. We have to prove that there are K', σ'' for which $\sigma = val\big(K',\sigma''|\mathcal{U}_1\big)(\sigma'')$, where $K',\sigma'' \models \bigwedge C_1$. Because path conditions do not contain program variables, from $K,\sigma' \models \bigwedge C_2$ it follows that this holds for *any* state, in particular, $K,\sigma \models \bigwedge C_2$. From the definition of $t^{=}_{\mathcal{U}_2}$ we obtain that furthermore $K,\sigma \models t^{=}_{\mathcal{U}_2}$ (as $\sigma = val\big(K,\sigma'|\mathcal{U}_2\big)(\sigma')$, cf. Def. 5.13). This allows us, together with the semantic entailment derived from the proof's premise, to deduce $K,\sigma \models \theta\left(\bigwedge C_1 \wedge t^{=}_{\mathcal{U}_1}\right)$. Choose K' such that it interprets symbolic values according to the substitution θ and otherwise like K. Therefore, $K',\sigma \models t^{=}_{\mathcal{U}_1}$ and, for any state σ''' (since C_1 does not contain program variables), $K',\sigma''' \models \bigwedge C_1$. Moreover, from $K',\sigma \models t^{=}_{\mathcal{U}_1}$ it follows by definition of $t^{=}_{\mathcal{U}_1}$ that there is a σ'' for which it holds that $\sigma = val\big(K',\sigma''|\mathcal{U}_1\big)(\sigma'')$, and because of $K',\sigma'' \models \bigwedge C_1$, it is true that $\sigma \in \bigcup_{K,\sigma} \{val(K,\sigma|\mathcal{U}_1)(\sigma) \mid K,\sigma \models \bigwedge C_1\}$. □

We define a strong version of the JavaDL-backed subsumption checker.

Definition 5.14 (Strong JavaDL-Backed Subsumption Checker)**.** Let $s_1 = (C_1, \mathcal{U}_1)$, $s_2 = (C_2, \mathcal{U}_2)$ be two SESs. We define the relation $\rhd^? \in \mathbb{S}_{SE} \times \mathbb{S}_{SE}$ s.t. $s_1 \rhd^? s_2$ iff we can derive in the JavaDL calculus that

$$\vdash \exists \overrightarrow{v_2};\left(C_2[\overrightarrow{v_2}/fpv(C_2)] \wedge t^{=}_{\mathcal{U}'_2}\right) \rightarrow \exists \overrightarrow{v_1};\left(C_1[\overrightarrow{v_1}/fpv(C_1)] \wedge t^{=}_{\mathcal{U}'_1}\right)$$

where $\overrightarrow{v_i}$ is a tuple of fresh logic variables matching in types and arity the tuple of free program variables $fpv(C_i)$ of path condition C_i, and $t^{=}_{\mathcal{U}'_i}$ is a clash-free translation of update $\mathcal{U}'_i := (fpv(C_i) := \overrightarrow{v_i}) \circ \mathcal{U}_i$ into a conjunction of equations, such that $K,\sigma,\beta \models t^{=}_{\mathcal{U}'_i}$ iff there is a $\sigma' \in \mathcal{S}$ s.t. $\sigma = val\big(K,\sigma',\beta|\mathcal{U}'_i\big)(\sigma')$. ◊

Lemma 5.9 (The Strong JavaDL-Backed Subsumption Checker is Correct)**.** *The relation* $\rhd^? \subseteq \mathbb{S}_{SE} \times \mathbb{S}_{SE}$ *defined in Def. 5.14 satisfies Def. 5.12.*

Proof. We have to show that

$$(C_1, \mathcal{U}_1) \rhd^? (C_2, \mathcal{U}_2) \text{ implies } (C_1, \mathcal{U}_1) \rhd (C_2, \mathcal{U}_2).$$

Let C_2', $\mathcal{U}_2'$, C_1', $\mathcal{U}_1'$, $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ be as in Def. 5.14. Assume that there is a derivation of $\vdash \exists \overrightarrow{v_2}; \left(C_2' \wedge t^{=}_{\mathcal{U}_2'}\right) \rightarrow \exists \overrightarrow{v_1}; \left(C_1' \wedge t^{=}_{\mathcal{U}_1'}\right)$. Since the JavaDL calculus is sound, the entailment

$$\exists \overrightarrow{v_2}; \left(C_2' \wedge t^{=}_{\mathcal{U}_2'}\right) \models \exists \overrightarrow{v_1}; \left(C_1' \wedge t^{=}_{\mathcal{U}_1'}\right) \tag{$\star$}$$

holds. Let K, σ be an arbitrary structure and state. If

$$\bigcup_{\beta} \left\{ val\left(K, \sigma, \beta | \mathcal{U}_2'\right)(\sigma) \mid K, \sigma, \beta \models \bigwedge C_2' \right\} = \emptyset,$$

we are done; therefore, assume that there is a state σ' and a variable assignment β s.t. $K, \sigma, \beta \models \bigwedge C_2'$ and $\sigma' = val\left(K, \sigma, \beta | \mathcal{U}_2'\right)(\sigma)$. We have to prove that there is a variable assignment β' for which $\sigma' = val\left(K, \sigma, \beta' | \mathcal{U}_1'\right)(\sigma)$, where $K, \sigma, \beta' \models \bigwedge C_1'$. Because path conditions do not contain program variables, from $K, \sigma, \beta \models \bigwedge C_2'$ it follows that this holds for *any* state, in particular, $K, \sigma', \beta \models \bigwedge C_2'$. From the definition of $t^{=}_{\mathcal{U}_2'}$ we obtain that furthermore $K, \sigma', \beta \models t^{=}_{\mathcal{U}_2'}$ (as $\sigma' = val\left(K, \sigma, \beta | \mathcal{U}_2'\right)(\sigma)$, cf. Def. 5.14). The left-hand side of ($\star$) holds in K, σ' if we choose $\overrightarrow{v_2}$ according to β. Therefore, there is a tuple $\overrightarrow{v_1}$ for which the right-hand side is true; select β' accordingly. We conclude $K, \sigma', \beta' \models C_1' \wedge t^{=}_{\mathcal{U}_1'}$. Since C_1' does not contain program variables, it holds that $K, \sigma, \beta' \models C_1'$. Moreover,

$$\begin{aligned} & \sigma' = val\left(K, \sigma, \beta | \mathcal{U}_2'\right)(\sigma) \;\wedge\; K, \sigma', \beta \models t^{=}_{\mathcal{U}_2'} \;\wedge\; K, \sigma', \beta' \models t^{=}_{\mathcal{U}_1'} \\ \Longrightarrow\; & \sigma' = val\left(K, \sigma, \beta' | \mathcal{U}_1'\right)\sigma \end{aligned}$$

and thus $\sigma' \in \bigcup_{\beta} \left\{ val\left(K, \sigma, \beta | \mathcal{U}_1'\right)(\sigma) \mid K, \sigma, \beta \models \bigwedge C_1' \right\}$. □

5.4.5 Soundness, (In)completeness, and Examples

Soundness, i.e., the property that only valid statements can be derived, and *completeness*, i.e., the property that all valid statements also can be derived, is in STL defined as usual for sequent calculi. Soundness can be shown by proving the soundness of all *rules*: A rule is sound if the validity of the conclusion follows from the validity of all premises. Then, we can conclude $\| \models \varpi$ from $\vdash \varpi$. A rule for which the reverse direction holds (validity of the premises follows from validity of the conclusion) is called complete. However, from the completeness of all rules, we cannot derive the completeness of our system; in particular, there is no incomplete rule in a system with *no* rules, which is obviously incomplete. Also, the existence of an incomplete rule does not automatically imply that the whole system is incomplete (whereas the existence of one unsound rule already destroys soundness of the whole system). Here, we prove soundness for all rules, from which the soundness of the system follows, and completeness for those rules which are complete. Our calculus is incomplete since for some language elements of symbolic traces, such as sequential composition, there are only incomplete rules.

Recall the definition of soundness and completeness of sequent calculus rules (stated earlier in Def. 2.10):

Definition 5.15 (Soundness and Completeness of Sequent Calculus Rules)**.** A rule

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2}{\Gamma \vdash \Delta}$$

of a sequent calculus is called

- *sound* if, whenever $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$ are universally valid, so is $\Gamma \vdash \Delta$.
- *complete* if, whenever $\Gamma \vdash \Delta$ is universally valid, then also $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$ are universally valid.

For rules with a different number of preconditions (non-branching or more than two branches) and with side conditions, the requirements have to be modified accordingly. $\Diamond$

The following lemma defines a practical, sufficient condition for proving soundness and completeness of STL calculus rules at once.

Lemma 5.10 (Soundness and Completeness of STL Rules)**.** *Let $\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n$ be the premises and $\Gamma \vdash \Delta$ be the conclusion of an STL calculus rule. The rule is* sound and complete *if for all K, σ it holds that $tval(K, \sigma|\Gamma_1 \vdash \Delta_1) \cap \ldots \cap tval(K, \sigma|\Gamma_n \vdash \Delta_n) = tval(K, \sigma|\Gamma \vdash \Delta)$.*

Proof. We may assume

$$tval(K, \sigma|\Gamma_1 \vdash \Delta_1) \cap \cdots \cap tval(K, \sigma|\Gamma_n \vdash \Delta_n) = tval(K, \sigma|\Gamma \vdash \Delta)\,. \tag{5.6}$$

Soundness: Assume that $\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n$ are universally valid. Then, for each

$i = 1, \ldots, n$, it holds for all K, σ that $tval(K, \sigma | \Gamma_i \vdash \Delta_i) = Traces$. Then, however, also the for the intersection we have $tval(K, \sigma|\Gamma_1 \vdash \Delta_1) \cap \cdots \cap tval(K, \sigma|\Gamma_n \vdash \Delta_n) = Traces$. Due to Eq. (5.6), this equals $tval(K, \sigma|\Gamma \vdash \Delta)$; it follows that $K, \sigma \Vert\!\models \Gamma \vdash \Delta$, and thus $\Vert\!\models \Gamma \vdash \Delta$.

Completeness: Assume that $\Gamma \vdash \Delta$ is universally valid; therefore, for all K, σ, it holds that $tval(K, \sigma|\Gamma \vdash \Delta) = Traces$. Due to Eq. (5.6), we have

$$tval(K, \sigma|tval(K, \sigma|\Gamma_1 \vdash \Delta_1) \cap \cdots \cap tval(K, \sigma|\Gamma_n \vdash \Delta_n)) = Traces,$$

which implies, for each $i = 1, \ldots, n$, $tval(K, \sigma|\Gamma_n \vdash \Delta_n) = Traces$, and thus $\Vert\!\models \Gamma_n \vdash \Delta_n$. □

The condition of Lem. 5.10 is not *necessary* since it requires that the *same* structure and state satisfy all premises and the conclusion of a rule, which is too strict. It is, however, useful since it allows to reduce soundness *and* completeness of a rule to a single equation. Furthermore, the condition is even necessary for (sound and complete) rules which do not introduce new program variables and rigid symbols, which applies to all our rules.

Theorem 5.11. *All STL calculus rules are sound; rules* not *labeled with "(!)" are complete.*

Proof Sketch. The proofs for complete, non-closing rules work by Lem. 5.10. The incomplete rules can be proven *sound* because the context is removed in the premises (one of the aspects causing their incompleteness). Rule closeSubsume is sound and complete since the definition of strong subsumption mirrors that of the semantics of SESs; rules close and close$\top^\infty$ are trivial. Rule closeUnsat is correct because by the side condition, the set $tval(K, \sigma|(C, \mathcal{U}))$ is empty, thus the sequent is valid. Details are in Appendix D. □

We inspect two example applications of STL. In the first one, we construct a nontrivial abstraction for a bug fixing scenario; the resulting proof obligation, on the other hand, is trivial. In the second example, we look into the verification of a temporal property.

Example 5.11 (STL Proof for Bug Fixing Scenario)**.** We return to Example 5.5 showing a "bug fixing" scenario. A program $p_{buggy} :=$ `if (x<-1) {x=-x;}` should compute the absolute of `x`. However, the guard of the `if` statement is wrong, thus a value of -1 is not inverted. We want to prove in STL that $p_{fixed} :=$ `if (x<=-1) {x=-x;}` is a correct fix. First, we compute the symbolic traces of the programs:

$$\begin{aligned} p_{buggy}{}^s &= (\text{true}); (((\mathsf{x} < -1); (\mathsf{x} < -1, \mathsf{x} := -\mathsf{x})) + (\mathsf{x} \geq -1)) \\ p_{fixed}{}^s &= (\text{true}); (((\mathsf{x} \leq -1); (\mathsf{x} \leq -1, \mathsf{x} := -\mathsf{x})) + (\mathsf{x} > -1)) \end{aligned}$$

Recall the patch abstraction α_{patch}, which adds the trace set

$$\mathcal{T}_{patch} \coloneqq \{\sigma\sigma[\mathsf{x} \mapsto -\sigma(\mathsf{x})] \mid \sigma \in \mathcal{S} \wedge \sigma(\mathsf{x}) = -1\}$$

The symbolic patch abstraction $\mathfrak{a}_{patch}$ can be defined as

$$\mathfrak{a}_{patch}(\varpi) \coloneqq ((\neg\mathsf{x} \doteq -1) \bowtie \varpi) + \big((\mathsf{x} \doteq -1); \big(\mathsf{x} \doteq -1, \mathsf{x} := -\mathsf{x}\big)\big)$$

By applying $\mathfrak{a}_{patch}$ to ${p_{buggy}}^s$, we add to all states in ${p_{buggy}}^s$ the assumption that x is not -1, and append the new two-step trace for the case where x *is* -1:

$$\begin{aligned}\mathfrak{a}_{patch}({p_{buggy}}^s) = (\{\text{true}, \neg\mathsf{x} \doteq -1\}); \Big(\big((\{\mathsf{x} < -1, \neg\mathsf{x} \doteq -1\}); \big(\{\mathsf{x} < -1, \neg\mathsf{x} \doteq -1\}, \mathsf{x} := -\mathsf{x}\big)\big) \\ + (\{\mathsf{x} \geq -1, \neg\mathsf{x} \doteq -1\})\Big) + \big((\mathsf{x} \doteq -1); \big(\mathsf{x} \doteq -1, \mathsf{x} := -\mathsf{x}\big)\big)\end{aligned}$$

where $(\{\text{true}, \neg\mathsf{x} \doteq -1\})$ can be simplified to $(\neg\mathsf{x} \doteq -1)$, $(\{\mathsf{x} < -1, \neg\mathsf{x} \doteq -1\})$ to $(\mathsf{x} < -1)$, and $(\{\mathsf{x} \geq -1, \neg\mathsf{x} \doteq -1\})$ to $(\mathsf{x} > -1)$. Big-step abstraction $\mathfrak{a}^{fin}_{big}$ replaces all intermediate states by a single "$\top$". Applying $\mathfrak{a}_{patch}$ and $\mathfrak{a}^{fin}_{big}$ therefore yields (after simplification)

$$\begin{aligned}(\mathfrak{a}_{patch} \circ \mathfrak{a}^{fin}_{big})({p_{buggy}}^s) = {} & \big((\neg\mathsf{x} \doteq -1); \top; (((\mathsf{x} < -1, \mathsf{x} := -\mathsf{x})) + (\mathsf{x} > -1))\big) + \\ & \big((\mathsf{x} \doteq -1); \top; \big(\mathsf{x} \doteq -1, \mathsf{x} := -\mathsf{x}\big)\big) \\ (\mathfrak{a}_{patch} \circ \mathfrak{a}^{fin}_{big})({p_{fixed}}^s) = {} & \big((\neg\mathsf{x} \doteq -1); \top; (((\mathsf{x} < -1, \mathsf{x} := -\mathsf{x})) + (\mathsf{x} > -1))\big) + \\ & \big((\mathsf{x} \doteq -1); \top; \big(\mathsf{x} \doteq -1, \mathsf{x} := -\mathsf{x}\big)\big)\end{aligned}$$

Those traces are literally equal, therefore, we can prove

$$\models \langle p_{fixed} \Vdash_{\alpha_{patch} \circ \alpha^{fin}_{big}} p_{buggy} \rangle$$

by an STL proof of $\vdash \neg(\neg(\mathfrak{a}_{patch} \circ \mathfrak{a}^{fin}_{big})({p_{buggy}}^s) + (\mathfrak{a}_{patch} \circ \mathfrak{a}^{fin}_{big})({p_{fixed}}^s))$ in one application of $\neg$right, one of $+$left, one of $\neg$left, and one of close. $\Diamond$

Example 5.12 (STL Verification of Temporal Logic Property). Consider the LTL specification $\bigcirc\big((\mathsf{done} \doteq \text{FALSE})^{ltl} \mathsf{U} \Box(\mathsf{x} \geq 0)^{ltl}\big)$ expressing that from the next state on, it must eventually be the case that x is always nonnegative, and until then, done must be FALSE. The corresponding symbolic trace is $(\text{true}); (\mathsf{done} \doteq \text{FALSE})^*; (\mathsf{x} \geq 0)^\omega$. We prove in STL that the following program satisfies this specification for the identity abstraction:

```
done = false;
if (x < 0) {
    /*@ loop_invariant x <= 0;
      @ decreases -x; */
    while (x < 0) {
      x += 1;
    }
}
done = true;
```

The symbolic trace (with invariant abstraction) for this program is

$$\begin{aligned}&(\text{true});(\emptyset, \mathsf{done} := \text{FALSE});\big((\mathsf{x} \geq 0, \mathsf{done} := \text{FALSE}) + \\ &\quad\big((\mathsf{x} < 0, \mathsf{done} := \text{FALSE});(\{\mathsf{x} < 0, c \leq 0\}, \mathsf{done} := \text{FALSE} \mathbin{||} \mathsf{x} := c)^*; \\ &\qquad(\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{done} := \text{FALSE} \mathbin{||} \mathsf{x} := c); \\ &\qquad\qquad(\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{x} := c \mathbin{||} \mathsf{done} := \text{TRUE})\big)\big)\end{aligned}$$

Note that we need the **decreases** term (loop variant) because otherwise, we could not show that eventually, x would be nonnegative; practically, the expression

$$(\{\mathsf{x} < 0, c \leq 0\}, \mathsf{done} := \text{FALSE} \mathbin{||} \mathsf{x} := c)^*$$

representing the loop would be $(\ldots)^\omega$ instead, which is not subsumed by $(\mathsf{done} \doteq \text{FALSE})^*$.

The STL proof tree for the problem is shown in Fig. 5.5. We use centered dots "$\cdots$" to abbreviate long symbolic traces; bottom-aligned dots "$\ldots$" represent abbreviated premises—which anyway are removed soon by elim; and $\mathsf{elim}^*{}_2$ applications.

As an example for subsumption checking, we show the proof obligation that the strong JavaDL-backed subsumption checker creates for the problem

$$(\mathsf{x} \geq 0) \rhd^? (\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{x} := c \mathbin{||} \mathsf{done} := \text{TRUE})$$

Observe that in the path conditions of both states, exactly x occurs freely, and is replaced by a fresh logic variable x_i. In both states, we prepend "$(\mathsf{x} := x_i)\circ\ldots$" to the symbolic store. For the right state, $(\mathsf{x} := x_2)\circ(\mathsf{x} := c \mathbin{||} \mathsf{done} := \text{TRUE})$ simplifies to $\mathsf{x} := c \mathbin{||} \mathsf{done} := \text{TRUE}$, since x is overwritten and does not occur as right-hand side in the store. The complete proof obligation is

$$\vdash \exists x_1;\ \big((x_1 < 0 \land c \leq 0 \land c \geq 0) \land (\mathsf{x} \doteq c \land \mathsf{done} \doteq \mathrm{TRUE})\big) \rightarrow \exists x_2;\ \big(x_2 \geq 0 \land \mathsf{x} \doteq x_2\big)$$

which is valid: From $c \leq 0 \land c \geq 0$ it follows that $c \doteq 0$, which together with $\mathsf{x} \doteq c$ implies $\exists x_2; \big(x_2 \geq 0 \land \mathsf{x} \doteq x_2\big)$ if we instantiate x_2 to c. $\Diamond$

5.5 Summary and Discussion

We presented Modal Trace Logic, a semantic framework based on traces, and Symbolic Trace Logic, a logic for reasoning about inclusion of symbolic traces. MTL is a "plug-in" logic: It only has one fixed syntactic component, the *trace modality* $[\mathscr{C}_{impl} \Vdash_\alpha \mathscr{C}_{spec}]$, which expresses that the *specification* $\mathscr{C}_{spec}$ approximates the *implementation* $\mathscr{C}_{impl}$ after the abstraction step defined by the *trace abstraction* α. MTL can be instantiated to concrete verification problems by integrating formal languages describing trace sets, so-called Trace Description Languages. Formally, this means that the *trace valuation function tval* has to be defined for expressions of the TDL ("TDL constructs"). For instance, the trace semantics for a TDL of deterministic sequential programs is defined, for every initial state σ, as the single trace resulting from the execution of a program starting in σ.

We defined several example TDLs and trace abstractions as well as a diamond version of the trace modality. MTL satisfies the two most important axioms of modal logic; however, in the case of axiom **K** only for some, yet common, combinations of abstractions and specification TDLs. An attempt to emulate PDL in MTL shows that the axiomatic system of PDL does not adequately describe semantics of the trace modality. For instance, trace modality expressions cannot be "flattened" (linearization). These insights are based on an instantiation of MTL to PDL *programs* only. Alternatively, we could have given the whole logic a trace semantics, including modalities $[\alpha]\varphi$. However, we aimed at characterizing the trace modality, and not to represent PDL formulas.

To demonstrate the versatility of MTL, we defined several program verification problems using the trace modality. We use different (combinations of) trace abstractions and TDLs, address functional as well as relational problems, and represent problems such as program synthesis and compilation using abstract programs. Since trace abstractions can be composed, we can construct many abstractions from a small set of reusable building blocks. Our notion of trace abstraction is well-behaving (e.g., gives rise to a Galois connection), albeit general enough to be applicable to interesting special cases. For example, we define

$$
\text{pull}^{\omega}\,\dfrac{\text{elim;}\,\dfrac{\text{closeSubsume}\,\dfrac{*}{(\mathsf{x} \geq 0) \vdash (\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{done} := \mathrm{FALSE} \,||\, \mathsf{x} := c)} \qquad \text{pull}^{\omega}\,\dfrac{\text{elim}^{\omega}{}_1\,\dfrac{\text{closeSubsume}\,\dfrac{*}{\ldots, (\mathsf{x} \geq 0) \vdash (\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{x} := c \,||\, \mathsf{done} := \mathrm{TRUE})}}{\ldots, (\mathsf{x} \geq 0); (\mathsf{x} \geq 0)^{\omega} \vdash (\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{x} := c \,||\, \mathsf{done} := \mathrm{TRUE})}}{(\mathsf{x} \geq 0)^{\omega} \vdash (\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{x} := c \,||\, \mathsf{done} := \mathrm{TRUE})}}{\ldots, (\mathsf{x} \geq 0); (\mathsf{x} \geq 0)^{\omega} \vdash (\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{done} := \mathrm{FALSE} \,||\, \mathsf{x} := c) \cdots}}{(\mathsf{x} \geq 0)^{\omega} \vdash (\{\mathsf{x} < 0, c \leq 0, c \geq 0\}, \mathsf{done} := \mathrm{FALSE} \,||\, \mathsf{x} := c) \cdots}
$$

$$(\star)$$

$$
\text{pull}^{*}\,\dfrac{\text{elim;}\,\dfrac{\text{closeSubsume}\,\dfrac{*}{(\mathsf{done} \doteq \mathrm{FALSE}) \vdash (\mathsf{x} < 0, \mathsf{done} := \mathrm{FALSE})} \qquad \text{elim}^{*}{}_2\,\dfrac{\text{closeSubsume}\,\dfrac{*}{(\mathsf{done} \doteq \mathrm{FALSE}) \vdash (\{\mathsf{x} < 0, c \leq 0\}, \mathsf{done} := \mathrm{FALSE} \,||\, \mathsf{x} := c)} \qquad (\star)}{(\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\{\mathsf{x} < 0, c \leq 0\}, \mathsf{done} := \mathrm{FALSE} \,||\, \mathsf{x} := c)^{*} \cdots}}{\ldots, (\mathsf{done} \doteq \mathrm{FALSE}); (\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\mathsf{x} < 0, \mathsf{done} := \mathrm{FALSE}); (\{\mathsf{x} < 0, c \leq 0\}, \mathsf{done} := \mathrm{FALSE} \,||\, \mathsf{x} := c)^{*} \cdots}}{(\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\mathsf{x} < 0, \mathsf{done} := \mathrm{FALSE}); (\{\mathsf{x} < 0, c \leq 0\}, \mathsf{done} := \mathrm{FALSE} \,||\, \mathsf{x} := c)^{*} \cdots}
$$

$$(\dagger)$$

$$
\text{¬right}\,\dfrac{\text{+right}\,\dfrac{\text{¬left}\,\dfrac{\text{elim;}\,\dfrac{\text{close}\,\dfrac{*}{(\mathrm{true}) \vdash (\mathrm{true})} \qquad \text{pull}^{*}\,\dfrac{\text{elim;}\,\dfrac{\text{closeSubsume}\,\dfrac{*}{(\mathsf{done} \doteq \mathrm{FALSE}) \vdash (\emptyset, \mathsf{done} := \mathrm{FALSE})} \qquad \text{+right}\,\dfrac{\text{elim}^{*}{}_1\,\dfrac{\text{pull}^{\omega}\,\dfrac{\text{elim}^{\omega}{}_1\,\dfrac{\text{closeSubsume}\,\dfrac{*}{\ldots, (\mathsf{x} \geq 0) \vdash (\mathsf{x} \geq 0, \mathsf{done} := \mathrm{FALSE})}}{\ldots, (\mathsf{x} \geq 0); (\mathsf{x} \geq 0)^{\omega} \vdash (\mathsf{x} \geq 0, \mathsf{done} := \mathrm{FALSE})}}{\ldots, (\mathsf{x} \geq 0)^{\omega} \vdash (\mathsf{x} \geq 0, \mathsf{done} := \mathrm{FALSE})}}{(\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\mathsf{x} \geq 0, \mathsf{done} := \mathrm{FALSE})} \qquad (\dagger)}{(\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\mathsf{x} \geq 0, \mathsf{done} := \mathrm{FALSE}) + ((\mathsf{x} < 0, \mathsf{done} := \mathrm{FALSE}) \cdots)}}{\ldots, (\mathsf{done} \doteq \mathrm{FALSE}); (\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\emptyset, \mathsf{done} := \mathrm{FALSE}); \cdots}}{(\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\emptyset, \mathsf{done} := \mathrm{FALSE}); \cdots}}{(\mathrm{true}); (\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega} \vdash (\mathrm{true}); (\emptyset, \mathsf{done} := \mathrm{FALSE}); \cdots}}{\neg((\mathrm{true}); (\emptyset, \mathsf{done} := \mathrm{FALSE}); \cdots), ((\mathrm{true}); (\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega}) \vdash}}{\neg((\mathrm{true}); (\emptyset, \mathsf{done} := \mathrm{FALSE}); \cdots) + ((\mathrm{true}); (\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega}) \vdash}}{\vdash \neg(\neg((\mathrm{true}); (\emptyset, \mathsf{done} := \mathrm{FALSE}); \cdots) + ((\mathrm{true}); (\mathsf{done} \doteq \mathrm{FALSE})^{*}; (\mathsf{x} \geq 0)^{\omega}))}
$$

Figure 5.5: Sequent Calculus Derivation for Example 5.12

a "patch abstraction" for bug fixing, which at the same time permits proving *relative* behavioral equivalence *and* semantically documents the patch applied to a buggy program.

As MTL does not have a static syntax, it is not possible to define a common reasoning system for all conceivable instantiations. To that end, we introduced STL, a logic for reasoning about inclusion of *regular symbolic traces*. STL is connected to MTL via a notion of sound and complete symbolic translations of TDL constructs. Its sequent calculus is sound, *but incomplete*. It is a legitimate question why an incomplete calculus should be preferred over the approach common in regular expression inclusion checking, which consists in using automata constructions and transformations: Construct NFAs for the implementation and specification, then create a Deterministic Finite Automaton (DFA) for the complement of the specification, and check whether the intersection with the implementation NFA is empty. This is not easy; It was shown in [MS72] that the inclusion problem for regular expressions is PSPACE-complete. Our problem is even more complicated: Atoms of our language consist of *symbolic states* and not atomic symbols. Therefore, creating the intersection is more involved, since edges of the two automata cannot be compared literally, but have to be checked for subsumption.

In [SH19b], we devised an automata-based solution using *Subsumption Simulation Relations (SSRs)* for checking inclusion, which saves the construction of intersections and negations. Symbolic traces are translated into *symbolic finite automata*; afterward, an attempt is started to create an SSR, using an external subsumption checker. The proposed algorithms are quite lengthy and complicated. Compared to that, it is fairly easy to find arguments for soundness and (in)completeness of the STL sequent calculus. We think that it therefore serves as a better basis to systematically explore the problem at hand. Moreover, the automata-based approach could not distinguish between finite and potentially infinite traces. The STL calculus does not have this restriction: In Example 5.12, we have to prove loop termination (specify a **`decreases`** term) since the specification requires that a property *eventually holds*, which it would not for nonterminating runs.

It would be interesting to see whether the calculus is complete (or can be completed) for interesting subclasses of the regular symbolic trace language. For regular expressions, there are results in this direction: Reference [Hov12] presents a polynomial-time algorithm for the restriction of the "specification" regular expression to "1-unambiguous" expressions, which can be computed incrementally with a one-symbol lookahead. One interesting problem class in our case is the restriction of the specification to expressions without choice (+); then, the incomplete rules will not discard potentially relevant premises in the antecedent. We leave these investigations to future work.

Finally, we would like to emphasize the value of symbolic traces and symbolic abstractions, which is independent of the chosen calculus for reasoning about them. Example 5.11, for instance, shows how a carefully chosen abstraction can even render the inclusion checking problem trivial. Moreover, the proposed "patch abstraction" concisely describes the effects of the applied patch: "For the problematic input of $x = -1$, do not execute the original program, but return the correct value 1 instead". We think that symbolic traces are an adequate and sufficiently general formalism for reasoning about MTL problems.

6 Correctness of Refactoring Techniques

Refactoring is the process of changing code in a way that does *not alter its external behavior*, yet improves its *internal structure* [Fow18]. This process is justified by the observation that designing a software system upfront (without writing code) and then coding it (without designing it) does not work well.[1] Programmers change their code frequently [MPB12], gradually increasing the gap between the initial design and the current state of the code: Engineering deteriorates to hacking. Refactoring, when exercised carefully and systematically, can contribute to maintainability and reusability of existing code.

It is also risky. When only regarding *functional* properties, one can even only lose, as refactoring consists in *changing working code* with the goal of *preserving its behavior*. Indeed, common refactorings can easily, and accidentally, change a program's behavior [EBS16]. Most refactorings come with preconditions and constraints. If those are not met, the transformed program might not compile, or—which is worse—compile, but behave wrongly under certain conditions. Frequently, refactorings are applied manually [MKF06], in which case the developer has to make sure that no constraints are violated. The standard precaution recommended to prevent the introduction of bugs by refactoring activities is *testing* [Fow99; Fow18]. This relies strongly on the quality of the existing test suite; when a test suite is insufficiently robust, testing may be misleading [AML17].

As many changes performed in software development comprise code refactorings (about 30% as reported in [Soa+11]), automatic support is indispensable. Indeed, many IDEs support common refactorings and automatic checking of required preconditions. This alone does not ensure correct results and can still lead to unexpected changes of a program's behavior [Dan+07; SGM13], as refactoring tools typically do not implement all preconditions [Soa+10]. Another line of work automatically generates test suites [Soa+10] or adds runtime assertions [EBS16] checking preservation of the original behavior.

There are three dimensions which are not covered by existing approaches:

(1) Although most applied refactorings are "low level" (64% according to [Soa+11]),

[1] Arguably, the same holds true for the process of writing a PhD thesis.

i.e., confined to method bodies, existing approaches almost exclusively address high-level refactorings such as “rename class / method”, “push down / pull up method”, “change signature”, “move class”, etc. The technique “rename variable” is an exception to this observation; also, we regard “extract method”, which is also considered in literature, as low level, even though it affects the class level, too.

(2) Existing work mostly focuses on enforcing the compliance with known preconditions and constraints of refactoring techniques, but does not examine whether these preconditions are sufficiently precise to exclude with certainty the introduction of undesired behavior. In other words, it is not systematically researched whether the lists of known constraints are correct and complete. Also here, there is an exception: Reference [GM06] proves correctness of three (high-level) refactoring techniques.

(3) Automatic creation of tests and runtime assertions is a practical and potentially efficient method for supporting developers. Notwithstanding, “heavyweight” formal verification of refactoring transformations can not only provide high confidence, but complete *certainty* about their correctness. We do not know of formal verification techniques specialized to program transformations by refactorings.

We developed REFINITY[2], a KeY-frontend for relational verification of abstract programs based on Abstract Execution. From Martin Fowler’s classic book [Fow99] and its second edition [Fow18], we chose nine statement-level refactoring techniques (six from the original book and three from the second edition), including two with loops. For each of the nine techniques, we created a REFINITY model consisting of two abstract programs: one representing the starting point, and one the result of the refactoring. In an iterative process, we refined the model by adding additional constraints, until we could prove *behavioral equivalence* of the abstract programs with the AE calculus discussed in Sect. 4.3. Thus, we obtain soundness of, for example, *Extract Method* at the same time as of its inverse, *Inline Method*. All proofs are fully mechanized in KeY and were conducted *fully automatically*. We chose refactorings at the statement level because they are directly expressible in JavaDL; moreover, this addresses the shortcoming described in Item (1). For each refactoring, we characterize the preconditions that make it semantics-preserving. Most preconditions are not mentioned in the literature. Thus, we tackle Item (2), on the one hand by elicitation of new constraints, and on the other hand by showing that conformance with them suffices to safely apply a refactoring technique. We also address Item (3): Since the semantics of abstract programs is defined by a set of JavaDL formulas (see Sect. 4.2), we can derive a set of formulas describing the constraints that a concrete program has to satisfy s.t. a given refactoring technique does not change the program’s

[2] `https://www.key-project.org/REFINITY/`

behavior. We do not have to verify that a whole method preserves its behavior. It suffices to instantiate the symbolic parts of the abstract program model by the relevant parts of the method body, and discard the arising proof obligations for the refactoring's constraints.

The chapter is structured as follows. We explain REFINITY, specifically the methodology to formalize refactorings as REFINITY models and the proof obligations that are generated from the models, in Sect. 6.1. In Sect. 6.2, we discuss our approach of using *abstract strongest loop invariants* to abstractly execute loops in REFINITY models. The refactoring preconditions we elicited are presented in Sect. 6.3. Sect. 6.4 concludes the chapter with a performance evaluation of AE / REFINITY and a short discussion.

6.1 Proving Refactorings with REFINITY

REFINITY is a frontend for KeY which allows to conveniently specify two abstract program models that can be related by common elements and relational pre- and postconditions. The tool has its own XML-based input format, and allows to specify the following elements of relational abstract program models:

- Two Java code fragments with Abstract Program Elements and AE constraints,
- A common method-level context,
- Program variables usable in both abstract programs without prior declaration,
- Dynamic frame specification variables (in other words "abstract location sets") that can be used in both abstract programs,
- Function and predicate symbols that can be used in both abstract programs,
- A list of "relevant locations" for the left program, and
- A list of "relevant locations" for the right program.

Relevant locations are—abstract or concrete—locations in which we are interested. The external behavior of the left and right program may differ in all locations which are provably disjoint from those that are declared relevant. As a standard, the lists of relevant locations comprise a single abstract location set "`relevant`". It is also possible to tag multiple location sets, program variables, or a mixture of both, as relevant.

Figure 6.1 contains a screenshot of the tool showcasing these elements. The "method-level context" is code that may appear inside the body of a class declaration. We use it to model the refactoring techniques *Extract Method*, *Decompose Conditional* and *Move Statements to Callers*. There is only one such context common to both programs.

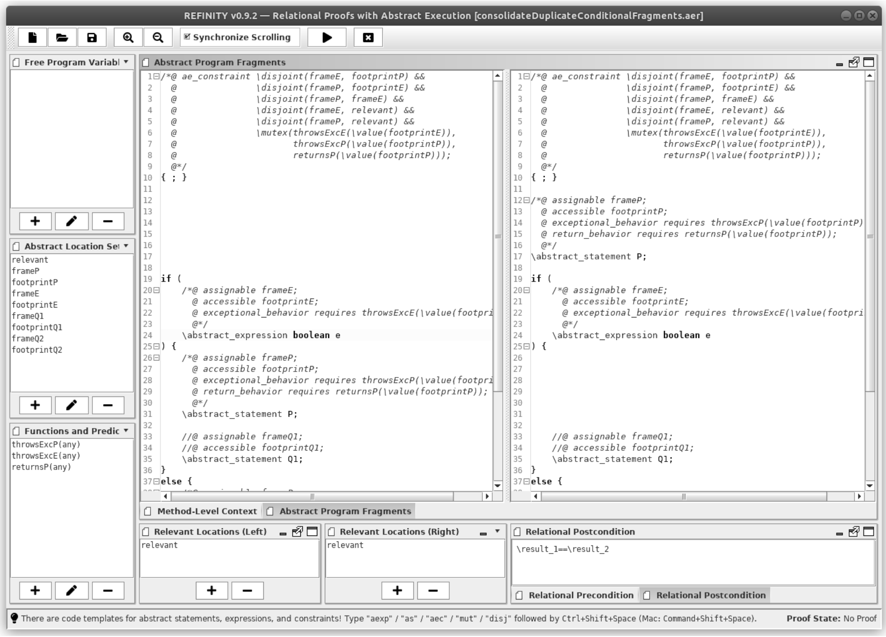


Figure 6.1: The REFINITY User Interface

We explain the workflow of proving a refactoring technique along a simple example: The *Slide Statements* refactoring [Fow18]. The refactoring consists in swapping two statements that appear in sequential order. The motivation for this refactoring is to keep related code together, thus increasing its understandability. Figure 6.2 shows a small example program before and after the refactoring. In the original version, the program first picks a random customer with a given name, then logs a purchase for the name, retrieves the address of the customer and sends a bill to that address. Logging is based on the name and not the retrieved customer object. Therefore, the second statement has a separate purpose than the other ones. By moving the statement to the front (and separating the two parts of the program by an empty line), we improve the code's readability.

```
Customer customer =
  customersFor(name).pickRandom();
logPurchase(name, amount);
String address = customer.getAddress();
sendBill(amount, address);
```

```
logPurchase(name, amount);

Customer customer =
  customersFor(name).pickRandom();
String address = customer.getAddress();
sendBill(amount, address);
```

Figure 6.2: Example Slide Statements Refactoring

By replacing concrete statements by Abstract Statements, we can prove not only the equivalence of the two versions of that individual (questionable) program, but the correctness of the *refactoring technique* itself, since the abstract program model can be specialized to all its legal instantiations. Listing 6.1 depicts an abstract program model for the original program before applying this refactoring; the transformed program contains the two ASs in reversed order. The postcondition specifies that the resulting state after program execution has to be the same for both sides in case of normal completion; if any of the ASs completed abruptly, then both sides have to return the same object or throw the same exception. This is in REFINITY formally captured by the following JML expression:

```
   ((returnsA(\value(footprintA)) ||
       returnsB(\value(footprintB))) &&
          \result_1[0] == \result_2[0])
|| ((throwsExcA(\value(footprintA)) ||
       throwsExcB(\value(footprintB))) &&
          \result_1[1] == \result_2[1])
|| \result_1==\result_2
```

The expressions **\result_1** and **\result_2** are untyped sequences that allow to access the post state of the left and right program, respectively. At position 0, returned objects are stored, and at position 1, there are thrown exceptions. Both are null if not present. From position 2 on, one can access the locations declared as "relevant", which for both sides is the abstract location set `relevant` in the present example. The abstract program model specifies that the frames of the two ASs are disjoint; furthermore, AS `A` may not write to the footprint of AS `B` and vice versa (Lines 6 to 8). Thus, the statements can be swapped without changing (in case of normal completion) the effects on the state, as neither AS can affect the other (write to the footprint of the other AS) nor overwrite its changes. Furthermore, we specified that the abrupt completion behavior of the statements is mutually exclusive (Lines 10 to 15). This is to prevent that the original program can complete for a different reason than the transformed program.

Since we declared as relevant location for both sides the abstract location set `relevant`, both programs have to terminate in *exactly identical* states. This is because we imposed no restrictions on `relevant`; therefore, *any* state change could affect its value and can therefore not be simplified away. The name "`relevant`" is not protected, we could have chosen a different name. The important principle is that it represents an arbitrary set of locations. In general, one can make different locations "relevant" (and consequently access them from positions 2 on in **`\result_1`** and **`\result_2`**). Furthermore, a location set `locset` can be declared *ir*relevant by specifying "**`\disjoint`**`(locset, relevant)`" in an **`ae_constraint`** delcaration. It then cannot affect the valuation of `relevant`.

Remark 6.1 (Behavioral Equivalence and Syntactic Aspects). With AE and REFINITY, we prove *behavioral equivalence* of programs. This does explicitly *not* comprise aspects like name binding and accessibility. For instance, it is not in the scope of our framework to assert that when extracting a method, the chosen method name is not yet bound in that class. This is in contrast to other approaches (e.g., [Sch+12; SSM15]), which *exclusively* address such aspects. We do not intend to extend our framework in that direction; also, KeY is not the right host system for such an endeavor (JavaDL *assumes* analyzed programs to be compilable without errors, and does not attempt to verify this). Instead, we suggest to *combine* our findings with syntactic aspects related to binding and accessibility discovered or enforced by other approaches. We also discuss this in Chapter 7. ◊

Methodology When specifying a new refactoring model, we start with two empty programs, relevant location set `relevant` and the relational postcondition

`\result_1==\result_2`.

Then, we specify the abstract programs with minimal annotations. Usually, each APE

Listing 6.1: Abstract Program Model for Slide Statements Refactoring

```
/*@ ae_specvars \locset relevant, frameA, footprintA, frameB, footprintB;
  @ ae_specvars \formula throwsExcA(any), throwsExcB(any),
  @                      returnsA(any), returnsB(any); */

/*@ ae_constraint
  @   \disjoint(frameA, frameB) &&
  @   \disjoint(frameA, footprintB) &&
  @   \disjoint(frameB, footprintA) &&
  @
  @   \mutex(returnsA(\value(footprintA)), returnsB(\value(footprintB))) &&
  @   \mutex(returnsA(\value(footprintA)),
  @          throwsExcB(\value(footprintB))) &&
  @   \mutex(throwsExcA(\value(footprintA)),
  @          throwsExcB(\value(footprintB))) &&
  @   \mutex(throwsExcA(\value(footprintA)), returnsB(\value(footprintB)));
  @*/

//@ assignable frameA;
//@ accessible footprintA;
//@ exceptional_behavior requires throwsExcA(\value(footprintA));
//@ return_behavior requires returnsA(\value(footprintA));
\abstract_statement A;

//@ assignable frameB;
//@ accessible footprintB;
//@ exceptional_behavior requires throwsExcB(\value(footprintB));
//@ return_behavior requires returnsB(\value(footprintB));
\abstract_statement B;
```

receives its own frame and footprint location set, but we begin without imposing constraints on them or on the abrupt completion behavior of APEs. Then, we initiate a proof attempt, which will usually fail at first. Inspecting the open goals provides information on how to refine the model to make the refactoring sound. Possible refinements include

(1) declaring the disjointness of abstract location sets,

(2) imposing mutual exclusion on abrupt completion behavior,

(3) declaring a functional postcondition for an APE, and

(4) refining the *relational* postcondition or

(5) adding a relational *pre*condition.

An alternative to this “top-down” approach is a “bottom-up” variant in which we declare a very restrictive initial model, disallowing any abrupt completion and specifying disjointness of all abstract location sets. Then, we can stepwise loosen the restrictions to obtain a more general model. This is most suitable for large and complicated models, as it helps to control the size of the arising proof trees and sequents. The caveat is that one might stop early in the generalization process, being satisfied with a closing proof; then, the result can be imprecise, in the sense that there exist more liberal refactoring preconditions that still allow for sound transformations. In particular for programs with loops, it can be sensible to first prove equivalence of the original program with *itself* to assert that the loops are correctly specified (see Sect. 6.2 for a discussion of loops).

Usually, one can choose between different refinements, leading to different models. A typical situation is when all open proof goals expect an APE to throw an exception. It is one possibility to forbid the APE to throw exceptions. Frequently, it is a better solution (leading to a more general model) to couple the exceptional behavior to an abstract precondition and declare mutual exclusion with the abrupt completion behavior of other APEs. Moreover, one can relax the relational postcondition—as we did for the *Slide Statements* example—such that in case of thrown exceptions, both sides have to throw equal exception objects, but the remaining state does not have to be equal. To derive sensible conclusions about the preconditions of a refactoring technique, there should be a convincing justification for each refinement that is applied.

REFINITY Models and Proof Obligations The “**`ae_specvars`**” keyword is not directly supported by KeY and REFINITY. Instead, REFINITY features input elements for the specification of abstract location sets and predicates (on the left in Fig. 6.1). The keywords “**`ae_constraint`**”, “**`\disjoint`**”, “**`\mutex`**” and “**`\value`**” are supported. Of those, all but "**\disjoint**" are new AE extensions. From the supplied specification elements, REFINITY creates a KeY problem file and a Java class `Problem` with two public methods `left` and `right`. The bodies of these methods are the abstract program fragments. Using self composition and Hoare triples, the problem specification could then be encoded as

$$\{\mathit{value}(\texttt{relevant}) \doteq \mathit{value}(\texttt{relevant'})\}$$
$$\texttt{Problem.left();Problem.right();}$$
$$\{\mathit{value}(\texttt{relevant}) \doteq \mathit{value}(\texttt{relevant'})\}$$

where locations in method `right` have to be renamed, similarly to `relevant'`, as common in self composition / product programs [BCK11]. More severely, abrupt completion of method `left` would lead to early termination of the whole program, skipping `right` completely. This could be mitigated by using **`try`** blocks. REFINITY follows a different approach not based on self-composition, which also avoids the need of renaming. Both programs are executed inside their own modality, accumulating results in different fresh predicates $\_P$ and $\_Q$ that each accept a sequence as parameter. An additional assumption then allows to connect the resulting expressions if the relational postcondition holds. In total, the created proof obligation has the following shape:

$$\begin{array}{ll}
 & \neg\texttt{obj} \doteq \mathit{null} \\
 & \land\ \mathit{exactInstance}_{\texttt{Problem}}(\texttt{obj}) \doteq \mathrm{TRUE} \\
 & \land\ \ldots \\
 & \land\ \mathit{Pre} \\
 & \land\ \{\texttt{_result} := \mathit{null} \,||\, \texttt{_exc} := \mathit{null}\} \\
 & \qquad \neg\langle \textbf{try}\ \texttt{\{ _result=obj.left()@Problem; \}} \\
 & \qquad\qquad \textbf{catch}\ \texttt{(Throwable t) \{ exc=t; \}}\rangle \\
 & \qquad\qquad\quad \neg_P(\texttt{_result}, \texttt{_exc}, \mathit{value}(\mathit{relevant})) \\
 & \land\ \{\texttt{_result} := \mathit{null} \,||\, \texttt{_exc} := \mathit{null}\} \\
 & \qquad \neg\langle \textbf{try}\ \texttt{\{ _result=obj.right()@Problem; \}} \\
 & \qquad\qquad \textbf{catch}\ \texttt{(Throwable t) \{ exc=t; \}}\rangle \\
 & \qquad\qquad\quad \neg_Q(\texttt{_result}, \texttt{_exc}, \mathit{value}(\mathit{relevant})) \\
\vdash & \quad \exists\, \mathit{Seq}\ _\mathit{res1};\ \exists\, \mathit{Seq}\ _\mathit{res2};\ \big(_P(_\mathit{res1}) \land _Q(_\mathit{res2}) \land \mathit{Post}(_\mathit{res1}, _\mathit{res2})\big)
\end{array}$$

The dots abbreviate further generic preconditions, such as the wellformedness predicate for the heap and disjointness of auxiliary variables, e.g., `_result`, with all abstract location sets declared in the model. "*Pre*" is an optional relational precondition formula. Free program variables specified in REFINITY are initialized to fresh values and passed as parameters to the methods `left` and `right` (we do not show this above). The third argument of $\_P$ and $\_Q$, $value(relevant)$, is replaced by the actually chosen relevant location(s) for the left and right program. Since the two predicates are fresh, the only way to prove them is by suitably instantiating the existentially quantified formula at the right of the sequent separator. This requires that the instantiations, which have to be the accumulated relevant values after method execution, satisfy the postcondition $Post(\_res1, \_res2)$, which can be any relation on the result sequences that can be expressed in JavaDL. This comprises more general (and complex) relations than simple equality. In the context of refactorings, equality of relevant locations, maybe in relaxed form to account for abrupt completion, is usually the desired relation. Occurrences of **`\result_1`** and **`\result_2`** are replaced by $\_res1$ and $\_res2$, respectively. As there is only one possibility for the instantiation of $\_seq1$ and $\_seq2$, the existential quantifier is no threat to automation. Indeed, all our refactoring models are proven fully automatically.

Understanding Failed Proof Attempts The implementation of AE in the semi-interactive theorem prover KeY has the advantage that we can *learn* from a failed proof attempt, since in case of a failed proof, the returned result is not simply "failed": Rather, it is possible to inspect the open proof tree and draw conclusions allowing to suitably refine the current model. We discuss some examples along the *Slide Statements* refactoring.

Too Liberal Frame Specifications When encountering an open goal of the shape

$$\begin{aligned}
& _P(null, null, \{\mathcal{U}_{\mathsf{A}}(frameA :\approx value(footprintA)) \mathbin{||} \\
& \quad \mathcal{U}_{\mathsf{B}}(frameB :\approx \{\mathcal{U}_{\mathsf{A}}(frameA :\approx value(footprintA))\}value(footprintB))\}), \Gamma \\
\vdash\; & _P(null, null, \{\mathcal{U}_{\mathsf{A}}(frameA :\approx value(footprintA)) \mathbin{||} \\
& \quad \mathcal{U}_{\mathsf{B}}(frameB :\approx value(footprintB))\}), \Delta
\end{aligned}$$

the declaration **`\disjoint`**`(frameA, footprintB)` helps to close the goal, since then, the inner application of the abstract update $\mathcal{U}_{\mathsf{A}}$ can be removed (by rule $\mathsf{dropUpdate}_5$, see Sect. 4.3). In the shown sequent, which arose from removing Line 7 in Listing 6.1, the antecedent formula contains the results for the left, and succedent formula for the right program. It suffices to look for occurrences of either *_P or _Q* terms; they contain the same information, only in reversed order (antecedent/succedent).

Missing Behavioral Constraints A situation like

$$\begin{aligned}
&_P(resultObjectA(\{\mathcal{U}_{\mathsf{A}}(frameA :\approx value(footprintA))\}footprintA),\\
&\qquad null, \{\mathcal{U}_{\mathsf{A}}(frameA :\approx value(footprintA))\}relevant),\\
&returns_A(value(footprintA)) \doteq \mathrm{TRUE},\\
&returns_B(value(footprintB)) \doteq \mathrm{TRUE},\ \Gamma\\
\vdash\ &_P(resultObjectB(\{\mathcal{U}_{\mathsf{B}}(frameB :\approx value(footprintB))\}footprintB),\\
&\qquad null, \{\mathcal{U}_{\mathsf{B}}(frameB :\approx value(footprintB))\}relevant),\ \Delta
\end{aligned}$$

shows that two ASs, A and B, both completed due to a **`return`** of a value, and that the returned objects and the effects on the state are not equivalent. This sequent was created by KeY after commenting out Line 10 in Listing 6.1, which specifies mutual exclusion on the return behavior of A and B. Indeed, if one of the statements returns a value, this value will generally be different to a value returned by the *other* statement; additionally, the returning statement can change the heap. The other statement has no chance to do so in one of the program versions. For some refactorings, declaring mutual exclusion is not enough: In the case of *Extract Method*, the extracted fragment must not return, since then, the effects of the **`return`** statement on the control flow (return from the outer vs. the extracted method) will be different. In those cases, the presence of a premise like $returns\_A(value(footprintA)) \doteq \mathrm{TRUE}$ indicates a problem. It can be resolved by adding the annotation "**`exceptional_behavior requires false`**`;`" to the APE.

6.2 Refactorings with Loops: Abstract Strongest Invariants

Symbolic Execution of loops requires advanced techniques. In (heavyweight) SE of individual programs, it is common to use *loop invariants* (cf. Sect. 2.3) to abstract from the concrete behavior of the loop. A loop invariant holds upon each entry to the loop, and good ones are strong enough to prove a postcondition. For simple postconditions, it suffices to find simple invariants. The situation is more complicated for relational verification of two programs, as we already briefly discussed in Sect. 5.4. We demonstrate this along an example from [BU18]. Assume a program `p(x)` operating on a single variable `x`. To prove the simple Hoare triple assertion $\{\mathtt{x} \doteq \mathtt{x}_c\}\ \mathtt{p(x);p(x}_c\mathtt{);}\ \{\mathtt{x} \doteq \mathtt{x}_c\}$, i.e., that `p` is equivalent *to itself*, we have to specify all loops in `p` with their *strongest possible loop invariants*, which, together with the negated guards and potentially further preconditions, are satisfied by exactly one value. For any weaker loop invariant leaving more freedom of choice for the values of the variables, the equality $\mathtt{x} \doteq \mathtt{x}_c$ cannot be shown.

It is easy to find a loop invariant, at least for partial completeness: "true" is a valid one for any loop. Finding good loop invariants, either manually or with automatic support, is difficult; finding *strongest* loop invariants is a hard problem. This problem can be solved by not executing each program in isolation, but to *interlock* the execution and use *coupling invariants* instead of (functional) loop invariants. For the example above, it suffices to show that after each execution of both loops, they have iterated equally often and $\mathtt{x} \doteq \mathtt{x}_c$ holds. Such simple coupling invariants can frequently be inferred automatically.

An elegant solution to reduce relational to functional verification and make use of coupling invariants is the idea of *product programs* [BCK11]. For structurally similar programs, one can, simply speaking, merge two loops in the factor programs to a single one in the product, and encode the coupling invariant as functional loop invariant. This, notwithstanding, does not work well with abruptly completing input programs. A remedy for this are *completion scopes* (Sect. 2.4), which allow to "catch" not only thrown exceptions, but also, e.g., **`return`**s and **`break`**s. This is nontrivial to realize and will likely result in code that is difficult to understand, which is why we turn back to functional invariants.

It turns out that while finding strongest loop invariants in the concrete case is very difficult, it is very *easy* in the abstract case. We simply declare an abstract predicate *Inv* and specify by an AE constraint that this is a strongest loop invariant for the loop at hand. We call *Inv* an *abstract strongest loop invariant*. This shifts the complexity to instantiation checking, as we have to supply an actual strongest invariant when testing whether a concrete program is an instance of the abstract model. For uncovering refactoring preconditions and proving the correctness of abstract programs, it is unproblematic. Moreover, we do not have to change anything in the already existing framework provided by REFINITY, and can still prove the equivalence of abstract programs with loops.

We first characterize strongest loop invariants. As stated before, a strongest loop invariant is, together with the negated loop guard, only satisfied by a single value. In slightly sloppy notation, we can formalize this as $\exists!\mathtt{x}; (\mathit{Inv}(\mathtt{x}) \land \neg g(\mathtt{x}))$, where *Inv* is a loop invariant and g the guard of a given loop, both operating on a single variable $\mathtt{x}$. We can rewrite the uniqueness quantifier $\exists!$ to $\exists v; \forall \mathtt{x}; ((\mathit{Inv}(\mathtt{x}) \land \neg g(\mathtt{x})) \leftrightarrow \mathtt{x} \doteq v)$.[3]

Strongest Loop Invariants for Abstract Programs To generalize this to abstract programs, assume that we have a loop with loop frame *loopFrame* and footprint *loopFootprint*, and that the result of the loop guard is bound to an abstract expression

[3] This formalization was inspired by the paper [DHS05], where a self composition-based information flow property is reduced to a JavaDL formula with quantifier shift, but only one occurrence of the input program.

$$guardIsTrue(value(loopFrame), value(loopFootprint)).$$

The formula

$$loopInv(value(loopFrame), value(loopFootprint))$$

is an *abstract strongest loop invariant* for the loop if it holds that

$$\exists\, Any\ \_fr, \_fp;\ \forall\, Any\, fr, fp;$$
$$\big((loopInv(fr,fp) \land \neg guardIsTrue(fr,fp)) \leftrightarrow \big(\_fr \doteq fr \land \_fp \doteq fp\big)\big)$$

This formula can be added to the relational precondition of a REFINITY model, or as an AE constraint to both abstract programs. Since the abstract predicate *loopInv* is uninterpreted, using it as an invariant will result in open goals for the *initially valid* and *preserved* cases. Thus, the invariant has to be explicitly established by one or a combination of several APEs. This can be specified by adding the invariant formula to their **`ensures`** clauses. In the case of *initially valid*, another option is to add an AE constraint in front of the loop, which makes sense if the loop is the first statement in the program fragment.

Listing 6.2 shows a fully specified abstract program containing a single loop with abstract guard and body for which we automatically can prove the equivalence to itself with REFINITY. In Lines 1 and 2, we declare the mentioned abstract specification variables for frame and footprint as well as the symbols for the abstract guard condition and loop invariant. The invariant is used in Line 15. To establish that instantiations of *loopInv* indeed are invariants for the given loop, we assert it initially in Lines 11 and 12 and demand by instantiations of AS `Body` that they also establish it (Line 29). Together with Lines 4 to 9, *loopInv* can only be instantiated by a strongest loop invariant.

Abrupt Completion The model shown in Listing 6.2 excludes abrupt completion of the APEs in the loop. The problem is that if we remove, for instance, the declaration "**`break_behavior requires false`**`;`", we cannot suitably instantiate the quantifier in Lines 4 to 9, as the guard is not necessarily false in case of abrupt completion due to a **`break`**. Therefore, we cannot show self-equivalence. In other words, we cannot infer that the left program completes abruptly in the same state as the right program, even though both programs are syntactically equal. The solution to this gives rise to a different, stronger notion of loop invariant: Instead of only requiring that the invariant holds upon each *entry* to the loop, we also require that it holds after each *abrupt exit* of the loop. In case of completion due to a **`break`**, we can replace the line

```
break_behavior requires false;
```

Listing 6.2: Abstract Strongest Loop Invariant for Partial Correctness

```
/*@ ae_specvars \locset loopFrame, loopFootprint;
  @ ae_specvars \formula guardIsTrue(any, any), loopInv(any, any); */

/*@ ae_constraint
  @   (\exists any _fr,_fp; (\forall any fr,fp; ((
  @         loopInv(fr, fp) &&
  @         !guardIsTrue(fr, fp)
  @       ) <==> (fr == _fr && fp == _fp))
  @   )); */

/*@ ae_constraint
  @   loopInv(\value(loopFrame), \value(loopFootprint)); */

/*@ loop_invariant
  @   loopInv(\value(loopFrame), \value(loopFootprint));
  @ assignable loopFrame;
  @*/
while (
    /*@ assignable \nothing;
      @ accessible loopFrame, loopFootprint;
      @ normal_behavior ensures \result <==>
      @   guardIsTrue(\value(loopFrame), \value(loopFootprint));
      @ exceptional_behavior requires false; */
    \abstract_expression boolean e
) {
    /*@ assignable loopFrame;
      @ accessible loopFootprint;
      @ normal_behavior ensures
      @   loopInv(\value(loopFrame), \value(loopFootprint));
      @ exceptional_behavior requires false;
      @ return_behavior requires false;
      @ break_behavior requires false;
      @ continue_behavior requires requires false;
    \abstract_statement Body;
}
```

by the lines

```
break_behavior ensures
  breaksBody(\value(loopFootprint)) &&
  loopInv(\value(loopFrame), \value(loopFootprint));
```

for an abstract predicate *breaksBody* and change the relational postcondition to

$$\exists\, Any\ \_fr, \_fp;\ \forall\, Any\ fr, fp;$$
$$\Big(\big(loopInv(fr,fp) \land \big(\neg guardIsTrue(fr,fp) \lor breaksBody(fp)\big)\big) \leftrightarrow \big(\_fr \doteq fr \land \_fp \doteq fp\big)\Big)$$

The updated model is shown in Listing 6.3, where abrupt completion of `Body` is allowed not only for **`break`**s, but also for **`return`**s, **`continue`**s and thrown exceptions. The changed lines are highlighted in gray. For abrupt completion due to a **`continue`**, we do not need an abstract predicate like *continuesBody*, since it suffices to establish the abstract loop invariant, as in the case of normal completion. The new model does still not allow abrupt completion of the *loop guard*; this is because generally, the loop guard alone will not be able to establish the "main" loop invariant.

We call this enhanced notion of abstract strongest loop invariants, which *also describe the loop's behavior in the case of abrupt completion*, "*strongest* abstract strongest loop invariants", for which we sometimes proudly use the less bulky wording "super invariants". For reference, we provide an overview of different loop invariant notions in Table 6.1. They are briefly described, and informally ordered by their "strength".

Total Correctness For *total* correctness, we have to prove that the loop indeed terminates. This is achieved in KeY and JavaDL by specifying a *loop variant*, which is usually an integer expression that is strictly decreased in every iteration of the loop, but never gets negative. Analogously to abstract invariants, we use *abstract variant terms* to this end. We declare a new function symbol $decrTerm : Any, Any \to int$ to represent the abstract variant. Listing 6.4 shows the abstract loop model, suitably specified for proving self-equivalence with termination. The loop invariant is enriched by the restriction that the variant is positive, and we add the abstract variant in the **`decreases`** clause of the loop. Instantiations of the loop body are required to establish the variant, i.e., decrease the valuation of the variant term, which, however, must not get negative. To specify this, we add a ghost variable `oldDecrTerm` before AS `Body` to remember the previous value of the variant.

Listing 6.3: Abstract Strongest Loop Invariant with Abrupt Completion

```
/*@ ae_specvars \locset loopFrame, loopFootprint;
  @ ae_specvars \formula guardIsTrue(any, any), loopInv(any, any);
  @ ae_specvars \formula throwsExcBody(any), returnsBody(any), breaksBody(any); */

/*@ ae_constraint
  @   (\exists any _fr,_fp; (\forall any fr,fp; ((
  @         loopInv(fr, fp) &&
  @         (   !guardIsTrue(fr, fp)
  @          || throwsExcBody(fp) || returnsBody(fp) || breaksBody(fp))
  @       ) <==> (fr == _fr && fp == _fp))
  @   )); */

/*@ ae_constraint
  @   loopInv(\value(loopFrame), \value(loopFootprint)); */
/*@ loop_invariant loopInv(\value(loopFrame), \value(loopFootprint));
  @ assignable loopFrame;
  @*/
while (
    /*@ assignable \nothing;
      @ accessible loopFrame, loopFootprint;
      @ normal_behavior ensures \result <==>
      @   guardIsTrue(\value(loopFrame), \value(loopFootprint));
      @ exceptional_behavior requires false; */
    \abstract_expression boolean e
) {
    /*@ assignable loopFrame;
      @ accessible loopFootprint;
      @ normal_behavior ensures
      @   loopInv(\value(loopFrame), \value(loopFootprint));
      @ exceptional_behavior ensures
      @   throwsExcBody(\value(loopFootprint)) &&
      @   loopInv(\value(loopFrame), \value(loopFootprint));
      @ return_behavior ensures
      @   returnsBody(\value(loopFootprint)) &&
      @   loopInv(\value(loopFrame), \value(loopFootprint));
      @ break_behavior ensures
      @   breaksBody(\value(loopFootprint)) &&
      @   loopInv(\value(loopFrame), \value(loopFootprint));
      @ continue_behavior requires ensures
      @   loopInv(\value(loopFrame), \value(loopFootprint)); */
    \abstract_statement Body;
}
```

Table 6.1: Different Notions of Loop Invariants

Concept	Description
Loop Invariant	A formula which holds whenever entering the loop.
Inductive Loop Invariant	A loop invariant which is sufficiently strong to prove the postcondition of the unit in which the loop appears.
Strongest Loop Invariant	A loop invariant which, together with a potential precondition and the negated loop guard, is only satisfied by a single value.
Abstract (Inductive/Strongest) Loop Invariant	(Inductive/Strongest) Loop invariant for a loop with abstract guard and/or body (containing ASs/AExps).
Strongest (Abstract) Strongest Loop Invariant	A strongest (abstract) loop invariant which also holds whenever the loop guard or body complete abruptly.

In the next section, we discuss the preconditions we found by stepwise refinement of refactoring models. We also show how to apply the principles for reasoning about abstract looping programs presented in this section.

6.3 Results: Preconditions for Statement-Level Refactorings

We created REFINITY models of nine statement-level refactoring techniques. *Slide Statements* [Fow18] was already discussed above; the refactoring *Consolidate Duplicate Conditional Fragments* [Fow99], which we treated in four versions, is a variant of this where a duplicated statement is pulled outside an **`if`** or **`try`** block. A similar refactoring, *Consolidate Conditional Expressions* [Fow99], merges several conditionals with the same bodies. Three refactoring techniques involve the definition of context containing additional method definitions, i.e., they are, strictly speaking, no "statement-level" refactorings: *Extract Method*, its variant *Decompose Conditional* [Fow99], and *Move Statements to Callers* [Fow18]. Technique *Replace Exception with Test* [Fow99] proposes a transformation of a **`try`**-**`catch`** block into an **`if`** statement. Finally, we modeled two refactorings with loops: *Split Loop* [Fow18] suggests splitting a loop performing two "different" tasks into two sequential loops. We

Listing 6.4: Abstract Strongest Loop Invariant for Total Correctness

```
/*@ ae_specvars ... @*/
/*@ ae_constraint
  @   (\exists any _fr,_fp; (\forall any fr,fp; ((
  @         loopInv(fr, fp) &&
  @         decrTerm(fr, fp) >= 0;
  @         (   !guardIsTrue(fr, fp)
  @          || throwsExcBody(fp) || returnsBody(fp) || breaksBody(fp))
  @       ) <==> (fr == _fr && fp == _fp)))); */

/*@ ae_constraint
  @   loopInv(\value(loopFrame), \value(loopFootprint)) &&
  @   decrTerm(\value(loopFrame), \value(loopFootprint)) >= 0; */

/*@ loop_invariant
  @   loopInv(\value(loopFrame), \value(loopFootprint)) &&
  @   decrTerm(\value(loopFrame), \value(loopFootprint)) >= 0;
  @ decreases decrTerm(\value(loopFrame), \value(loopFootprint));
  @ assignable loopFrame; */
while (/*@ assignable \nothing;
         @ accessible loopFrame, loopFootprint;
         @ normal_behavior ensures \result <==>
         @   guardIsTrue(\value(loopFrame), \value(loopFootprint));
         @ exceptional_behavior requires false; */
       \abstract_expression boolean e) {
    /*@ ghost int oldDecrTerm =
      @   decrTerm(\value(loopFrame), \value(loopFootprint)); */

    /*@ assignable loopFrame;
      @ accessible loopFootprint;
      @ normal_behavior ensures
      @   loopInv(\value(loopFrame), \value(loopFootprint)) &&
      @   decrTerm(\value(loopFrame), \value(loopFootprint)) >= 0 &&
      @   decrTerm(\value(loopFrame), \value(loopFootprint)) < oldDecrTerm;
      @ ... */
    \abstract_statement Body;
}
```

use two distinct abstract strongest loop invariants to model this. The refactoring *Remove Control Flag* [Fow99] consists in using a direct **break** instead of setting a flag inside a loop once a task is accomplished. In both loop-based refactoring techniques, loop bodies have to establish the loop invariant before completing abruptly.

There are only three refactorings with non-trivial preconditions: Two of the four variants of *Consolidate Duplicate Conditional Fragments* moving a common *postfix* to after a conditional or **try** block, and *Consolidate Conditional Expressions*. In that case, this is surprising, since Reference [Fow99] explicitly states "*if there are side effects, you won't be able to do this refactoring*", which is wrong.

Subsequently, we discuss all modeled refactoring techniques. We briefly introduce their motivation and basic mechanics, mainly quoting Fowler's books, and explain our results and interesting characteristics of the created models.

6.3.1 Slide Statements

The goal of *Slide Statements* [Fow18] is to reorder statements to keep those together which fulfill a common purpose. The mechanics are simple: Identify (1) the statement to move and (2) the target position, move the statement if there is no "inference" with the target position, and abort otherwise. Fowler quite precisely names the possible types of inference:

- A fragment cannot slide backward earlier than any element it references is declared,
- a fragment cannot slide forward beyond any element that references it,
- a fragment cannot slide over one that modifies an element it references, and
- a fragment that modifies an element cannot slide over any other element that references the modified element.

We did not find any further restrictions concerning read/write dependencies. However, our terminology allows to describe them more concisely: Let *frameA*, *footprintA*, *frameB* and *footprintB* be the frames and footprints of the involved statements `A` and `B`, respectively. Then, the following pairs have to be disjoint: (1) *frameA* and *frameB*, (2) *frameA* and *footprintB*, and (3) *frameB* and *footprintB*.

Our findings go beyond Fowler's descriptions w.r.t. abrupt completion (which [Fow99; Fow18] frequently do not discuss). To ensure equivalent behavior before and after the refactoring, neither of the involved statements must complete abruptly. Alternatively, it is possible to allow abrupt completion, but only *mutually exclusively*, i.e., `A` may complete abruptly if, and only if, B does *not* complete abruptly. Then, the refactoring is safe *if* neither of the swapped statements has *relevant* side effects in case of abrupt completion of the other one. Consider the program

```
... { int x = 17; throw new RuntimeException(); } ...
```

Swapping the assignment and the **throw** statement is safe: Since the variable x is declared locally in the block, it is reasonable to call it irrelevant. If x was a field or declared outside a **try** statement catching the thrown exception, applying the refactoring would be unsafe.

The complete code for the refactoring is in Appendix E, Fig. E.1 (Page 343).

6.3.2 Consolidate Duplicate Conditional Fragments

Consolidate Duplicate Conditional Fragments is a variant of *Slides Statements* proposed in the first edition of Fowler's book [Fow99]. The idea is to move code which is executed in all branches of a conditional to outside that conditional. This shortens the code and makes clearer what the conditional branches do differently. The mechanics are as follows:

- Identify code that is executed the same way regardless of the condition,
- if the common code is at the beginning, move it to before the conditional,
- if the common code is at the end, move it to after the conditional.

In addition, the possibility to slide code out of the middle of a conditional branch, which comes close to *Slide Statements*, and the variant of moving a statement out of a **try**-**catch** block are mentioned, but no additional preconditions for the variants with an **if** statement.[4] For the variant with **try** blocks, one can deduce from the descriptions the given precondition that the extracted statement should not throw an exception.

Figure 6.3 illustrates the four variants of the refactoring technique that we modeled in REFINITY. For the version with a **try**, Fowler talks about moving the postfix "to the final block", leaving unspecified whether this refers to the **finally** block or to the statements after the **try**. Therefore, we show two variants of extracting a postfix out of a **try** block (Figs. 6.3c and 6.3d). Extracting a postfix of an **if** statement (Fig. 6.3a) can be done without restrictions; the original and refactored version cannot even be distinguished by standard SE. For variant (c), no constraint applies but the one previously mentioned, that the extracted postfix must not throw an exception.

[4] Not even the usual recommendation to compile and test after each change is mentioned. This is probably due to the presumably innocuous character of expressions which, for instance, cannot declare variables. However, expressions may not only complete exceptionally, but also have side effects, such as in `i++>--j`.

```
if (expr) {
  Q1
  P
} else {
  Q2
  P
}
```

```
if (expr) {
  Q1
} else {
  Q2
}

P
```

(a) Extract Postfix

```
if (expr) {
  P
  Q1
} else {
  P
  Q2
}
```

```
P

if (expr) {
  Q1
} else {
  Q2
}
```

(b) Extract Prefix

```
try {
  Q1
  P
} catch
   (Throwable t) {
  Q2
  P
}
```

```
try {
  Q1
} catch
    (Throwable t) {
  Q2
}

P
```

(c) Extract Postfix (**try**, no **finally**)

```
try {
  Q1
  P
} catch
   (Throwable t) {
  Q2
  P
}
```

```
try {
  Q1
} catch
    (Throwable t) {
  Q2
} finally {
  P
}
```

(d) Extract Postfix (**try**, with **finally**)

Figure 6.3: Variants of Consolidate Duplicate Conditional Fragments. The **transformed** programs in Fig. 6.3c and Fig. 6.3d are not (unconditionally) equivalent to each other, although they result from the same source. For Fig. 6.3d, we have to additionally assume that Q_1 does not return.

Variant (d) additionally requires the leading statement in the **`try`** block, Q_1 in the figure, to not complete due to a **`return`**. Otherwise, the postfix P in the new **`finally`** block would be executed after, but not before the refactoring. In both variants (c) and (d), the extracted statement must not have access to the exception variable `t`.

The most interesting instance of the refactoring is variant (b). We discovered the following preconditions which are necessary for a safe application of the refactoring:

- The frame of P has to be disjoint from the footprint of *expr*. Otherwise, extracting P can influence the control flow, i.e., which branch of the conditional is taken. Similarly, the frame of *expr* has to be disjoint from the footprint of P.
- The *frames* of P and *expr* have to be disjoint. Otherwise, both APEs could overwrite effects caused by the other one.
- The frames of P and *expr* must not be relevant, i.e., disjoint from the set of relevant variables for both sides. For instance, if *expr* throws an exception, P has after the refactoring the chance to tamper with the state, but not before, similarly vice versa. Note that they are still relevant for the overall execution, as they can influence the computations of Q_1 and Q_2.
- Abrupt completion for *expr* and P has to be mutually exclusive; i.e., *expr* may only throw an exception if P does not return or throw an exception, etc. Otherwise, the programs could complete for different reasons before and after the refactoring.

There are no restrictions concerning Q_1 and Q_2.

The complete code for the refactoring is in Appendix E, Fig. E.2 (Page 344).

6.3.3 Consolidate Conditional Expression

When several checks in a series are different but have the same result, they can be consolidated into a single check connected by ands or ors. This makes the check clearer and frequently sets up for an application of *Extract Method*. The mechanics are described in [Fow99] as follows:

- Check that none of the conditionals has side effects; *if there are side effects, you won't be able to do this refactoring*,
- replace the string of conditionals with a single conditional statement using logical operators,
- compile and test.

```
if (expr1) {
  P
}

if (expr2) {
  P
}
```

```
if (expr1 ||
      expr2) {
    P
  }
}
```

(a) Consecutive Checks

```
if (expr1) {
  if (expr2) {
    P′
  }
}
```

```
if (expr1 &&
      expr2) {
    P′
  }
}
```

(b) Nested Checks

Figure 6.4: Variants of Consolidate Conditional Expressions. In Fig. 6.4a, P must either return or throw an exception.

We modeled two versions of this refactoring, one where two sequential **if** statements on the same level are merged using ors, and one where two nested **if** statements are merged using ands. They are shown in Figs. 6.4a and 6.4b. We interpreted "have the same result" such that the statement P must in each state either return or throw an exception, which is also the case for all examples in [Fow99]. We can loosen this restriction for the variant with nested **if** statements, where P' can be an arbitrary statement.

Both variants can be applied *without additional preconditions*, in contrast to the warning in [Fow99]. What is more, the two occurrences of P in Fig. 6.4a need not to return the "same result", only the same results for the same inputs, which is relevant if the expressions have side effects. The refactoring only is *not* safe with logical connectors *without short circuit evaluation*, i.e., "`|`" or "&", since then, side effects of $expr_2$ which were ineffective before the refactoring could be effective afterward.[5] In that case, it is true that only exception-free expressions without side effects may be used.

The complete code for the refactoring is in Appendix E, Figs. E.3 and E.4 (Pages 345 and 346).

6.3.4 Extract Method

When a method body is too long or not self-explanatory, one can extract code fragments that can be grouped together into a new method. The name of the extracted method then should describe its purpose well, improving clarity. Figure 6.5 is a schematic representation

[5] Reference [Fow99] targets Java, where operators "`||`" and "&&" are evaluated with short-circuit evaluation, i.e., later expressions that cannot change the boolean result of the check are not executed. The second edition [Fow18] describes the mechanics equivalently for JavaScript, which also has short-circuit evaluation.

of this technique. The mechanics are described in [Fow99] as follows (we omit some details which are not relevant for correctness):

- Create a new method, copy the extracted code from the source to the target method,
- choose as parameters for the new method the variables referenced in the extracted code which are in the local scope of the source method,
- remove parameters that are used in the extracted part only and instead declare them as temporary variables within the new method,
- abort if more than one of the parameters are assigned (as changes are not visible in the source method); treat the extracted code as a query if exactly one of the parameters are assigned (assign the result of the method to the variable concerned),
- compile,
- replace the extracted code in the source method with a call to the target method,
- compile and test.

$P(\texttt{var} :\approx args)$	`var = method(`*args*`);`
	
	`// ...`
	`Object method(`*args*`) {`
	    `Object var;`
	    $P(var :\approx args)$
	    return `var;`
	`}`

Figure 6.5: Extract Method Refactoring

Due to the semantics of abstract program fragments (cf. Def. 4.5), instances have to compile. Therefore, the restrictions related to the scopes of relevant variables are implicit in our models. Apart from that, we found two restrictions not mentioned in [Fow99] and [Fow18] (in the latter reference, the refactoring is called "*Extract Function*"):

- The extracted fragment *must not return*. This may seem obvious and is easily checkable, but is not mentioned in literature. A return from the extracted method has a different effect to the control flow than a return from the source method.
- If the extracted fragment *throws an exception*, it *must not change the value of the query result variable* `var` before. The change is visible before, but not after extraction, leading to different results if `var` is read by the code catching the exception.

The complete code for the refactoring is in Appendix E, Fig. E.5 (Page 347).

Remark 6.2 (Method Parameters). Our specification language for AE (see Sect. 4.1) has no notion of "abstract method parameters". Therefore, in the abstract program model for *Extract Method* (Fig. E.5), only the concrete program variable `var` is passed explicitly. This is, in fact, no restriction; all results also hold for arbitrary method parameters. It is only important whether they are in the instantiations of the abstract frames and footprints of the APEs in the method body. In the schema shown in Fig. 6.5, we could, for example, add a program variable `x` to the method `method`. If `x` is not in the frame and footprint of P, this does not make any difference; if it *is* in the frame or footprint of P, already discussed results also hold for the frame or footprint with `x`. Preconditions of refactorings with methods shown in this section apply for all extensions by method parameters and suitable instantiations of abstract frame specification variables. ◊

Remark 6.3 (Overapproximating Frames). Every APE only has one frame specification which has to overapproximate the frames for all behaviors. For instance, it is possible that an APE assigns different variables when it throws an exception and when it completes normally; one then has to define the union of all these locations as the general frame of the APE. If there were different frames for different behaviors, the restrictions concerning `var` from above would have to be generalized: If an APE completes for a reason *R*, it must not change locations specified for completion due to other reasons which are not in the frame specification for *R*. ◊

6.3.5 Decompose Conditional

The technique *Decompose Conditional* [Fow99] is a variant of *Extract Method*, in which the condition, then part, and else part of an **`if`** statement are extracted into separate methods with the intention to make complicated control flow better to understand. Since the refactoring consists of three applications of *Extract Method*, the same preconditions apply for the extracted *statements*. There is no precondition for the extracted *expression*.

We do not include the abstract program model since it does not provide new insights.

6.3.6 Move Statements to Callers

Move Statements to Callers is proposed in [Fow18] (its inverse is *Move Statements into Function*). The motivation for the refactoring is that during ongoing development, abstraction boundaries might shift. Then, the content of a method can lose its atomic character.

If the boundaries between caller and callee are too big, it is recommended to inline the method, apply *Slide Statements* and extract a new method. For small changes, however, it can suffice to simply move a subset of statements from a method to the callers.

The mechanics for this refactoring are described in [Fow18] as follows:

- *Either* cut the statements to move from the callee method and paste into the callers,
- *or* apply *Extract Method* to the statements *not* to move, apply *Inline Method* to the original method, and rename the extracted method to the original name.

```
res = methodBefore();

// ...
Object methodBefore() {
  A
  B
}
```

```
A
res = methodAfter();

// ...
Object methodAfter() {
  B
}
```

Figure 6.6: Move Statements to Callers Refactoring

Figure 6.6 illustrates this refactoring. The only restriction we found is that the moved statement *A* must not return, similar to *Extract Method*. As always, the code has to compile.

The complete code for the refactoring is in Appendix E, Fig. E.6 (Page 348).

6.3.7 Replace Exceptions with Test

Exceptions should be used for *unexpected* behavior, and not act as a substitute for conditional tests. The *Replace Exceptions with Test* refactoring technique [Fow99] proposes to introduce a check for a condition causing an exception when it is reasonable to expect that the condition can be checked. A good example is a division of two numbers put into a **`try`**-**`catch`** block since it is known that an `ArithmeticException` is raised if the divisor is zero. Instead, one should check beforehand whether the divisor is zero.

The mechanics for this refactoring are described in [Fow99] as follows:

- Put a test up front and copy the code from the **`catch`** block into the appropriate leg of the **`if`** statement,
- add an assertion to the **`catch`** block signalling whether the catch block is executed,
- compile and test,

- remove the **catch** block and the **try** block if there are no other **catch** blocks,
- compile and test.

```
try {                                             if (!(cond)) {
  P // throws exception if cond holds               P
} catch (Throwable t) {                           } else {
  Q                                                 Q
}                                                 }
```

Figure 6.7: Replace Exception with Test Refactoring

The schema for this refactoring is depicted in Fig. 6.7. A proof attempt for the technique unveiled the following problem: If statement P throws an exception, it might *change the relevant state before completing*, which it cannot do after the refactoring. We have proven that the refactoring is safe under each of the following conditions (alone):

(1) The frame of P is disjoint from the set of relevant locations and from the frame of Q.

(2) The frames of P and of Q are disjoint from the set of relevant locations, and Q always completes normally.

(3) The frame of P is disjoint from the footprint of Q, and Q has to assign all locations assigned by P.

(4) The statement Q starts with a "rollback", which resets all locations in the frame of P to some fixed values. These values must be independent from the frame of P, i.e., be constants or use locations disjoint from the locations written by P.

The first condition is simple: If the effects of P are not interesting at all, the refactoring is safe. In the second condition, P may influence Q, but only if both are not globally relevant, and if Q completes normally. The reason for the last restriction is that Q might return different results or throw different exceptions depending on the outcome of P. The third condition is less restrictive: If Q does not depend on the outcome of P and overwrites P's changes, the technique is sound. Finally, Item (4) probably does not apply to legacy code. However, it shows how to easily massage the code to make the refactoring safe. The ASs do not have to be changed, it suffices to add the block of "rollback" statements.

Appendix E contains a combined model for the first two conditions (Fig. E.7 on Page 349), one model for the third condition (Fig. E.8 on Page 350), as well as the code implementing the last condition (Fig. E.9 on Page 351). The models show how abstract predicates and postconditions can be used to bind different APEs together. We specify that the body of the **try** statement completes due to a thrown exception if, and only if, an abstract formula evaluates to true. The guard of the conditional is then specified such that it evaluates to true iff the negation of that abstract formula evaluates to true. This is a common pattern when developing abstract program models with interacting elements.

6.3.8 Split Loop

Split Loop [Fow18] is the first of two refactoring techniques addressing loops we studied. In essence, a loop performing two distinct things is split into two consecutive loops.

The mechanics for this refactoring are described in [Fow18] as follows:

- Copy the loop,
- identify and eliminate duplicate side effects, and test.

```
Init
/*@ loop_invariant
  @   loopInvG &&
  @   loopInvP &&
  @   loopInvQ; */
while (g) {
  P
  Q
}
```

```
Init
/*@ loop_invariant
  @   loopInvG &&
  @   loopInvP; */
while (g) {
  P
}

Init
/*@ loop_invariant
  @   loopInvG &&
  @   loopInvQ; */
while (g) {
  Q
}
```

Figure 6.8: Split Loop Refactoring

We modeled the refactoring in REFINITY with *abstract strongest invariants*. The schema is shown in Fig. 6.8. Each AS maintains its own invariant; in addition, there is an invariant for the guard, which could, for instance, impose a bound on the loop counter. After splitting, only the relevant invariant for the AS contained in the loop has to be maintained. This model gives rise to the following preconditions for a safe application of *Split Loop*:

- The frames of P and Q have to be disjoint from the footprint of g.
- The frames of P and Q have to be disjoint; also the frame of P has to be disjoint

from the footprint of Q and vice versa (P and Q have to be "independent").

- The guard g and statement P must not complete abruptly. Otherwise, they would have to establish the whole invariant, including unrelated parts.
- The statement Q may complete abruptly (because it appears last), but only if it also then establishes its invariant.

The invariant of the guard can either be established by the guard itself, which can have side effects ("`i++`"), or by an additional loop update statement. This statement would have to be added as first statement in the loop, otherwise, Q has to establish also the invariant of g when completing abruptly.

In particular in conjunction with abrupt completion, abstract strongest loop invariants are not only an effective instrument for proving relational properties about abstract programs with loops, but also a thinking tool. For instance, one can derive that it is generally bad if a loop guard throws an exception: Then, it will usually not be able to establish the whole invariant, leading to unexpected behavior. An exception occurring later in the loop is likely to be less harmful, if the code has been carefully designed.

The code for the refactoring is contained in Appendix E, Figs. E.10 to E.12 (Pages 352 to 354). We apply our results for this refactoring along an example.

Example 6.1 (Split Loop and Slide Statements)**.** We look at an example from [Fow18] computing the total salary and average age of the employees of a company:

```
People[] people = employees();
int avgAge = 0, totalSalary = 0, i = -1;
while (i < people.length) {
  i++;
  avgAge += people[i].age;
  totalSalary += people[i].salary;
}
avgAge = avgAge / people.length;
```

We instantiate the abstract program representing the original state before a *Split Loop* refactoring (Fig. E.10) to the example program, and show that the preconditions are met. This allows us to transform the program above according to the abstract model, at the same time obtaining the guarantee that this transformation is safe.

The loop is doing *two different things*: (1) Computing the *average age*. This task operates on the variable `avgAge` and reads the `age` field of the elements in the `people` array.

(2) Computing the *total salary*. This task writes to the variable `totalSalary` and reads from the `salary` field of the elements of the `people` array. Their loop invariants are

```
avgAge == (\sum int j; j >= 0 && j < i; people[j].age)
totalSalary == (\sum int j; j >= 0 && j < i; people[j].salary)
```

We already observe that those statements and invariants are independent: Their frames are disjoint, and neither statement writes to the accessed locations of the other.

The footprint of the loop guard are the variables `i` and the field `people.length`. Neither is written by one of the other statements. The invariant for the guard is

```
i >= -1 && i <= people.length
```

Together, the partial invariants are a strongest invariant for the loop: With the negated guard, we derive that $\mathtt{i} \doteq \mathtt{people.length}$, and thus `avgAge` and `totalSalary` contain the sums of fields `age` and `salary`, respectively, for all people from index 0 to (exclusively) `people.length`. To instantiate the refactoring, we have to assert that neither of the array accesses causes an exception. We need the additional assumption that the array is not **`null`** for that; the indices are all in range. Applying these instantiations to the abstract program model for the transformed program (Fig. E.11) leads to

```
1  People[] people = employees();
2  int avgAge = 0, totalSalary = 0; int i;
3
4  i = -1;
5  while (i < people.length) {
6    i++;
7    avgAge += people[i].age;
8  }
9  i = -1;
10 while (i < people.length) {
11   i++;
12   totalSalary += people[i].salary;
13 }
14
15 avgAge = avgAge / people.length;
16
```

Both loops still satisfy the invariant for the guard as well as the invariants associated to the statements in their body. We can now additionally apply *Slide Statements* to the swap the second loop (lines 9 to 13) with the final statement (line 15). This is possible according to our results from Sect. 6.3.1 since the frames and footprints of the statements are independent, and because furthermore, abrupt completion is mutually exclusive: There are no **`return`**s, and only the division by `people.length` can cause an exception.

For a final demonstration, let us replace Line 5 in the example program by

```
avgAge += people[i].age + (i >= people.length ? totalSalary : 0);
```

We can still apply the refactoring: Even though the variable `totalSalary` occurs in the statement, it is not in its (precise) footprint, since the expression `i >= people.length` is never true inside the loop. Only if we approximate the footprint by collecting literal occurrences of locations, the preconditions are not satisfied. In general, the syntax of instantiating programs is not important. We only care about their (big-step) *effects*. ◊

6.3.9 Remove Control Flag

In loops performing tasks such as searching through a data structure, one can frequently find a "control flag" which determines when the loop should stop looking (e.g., because the sought-after element has been found). The *Remove Control Flag* [Fow99] refactoring suggests to remove control flags and to replace them by **break** and **continue** statements, which makes the purpose of the conditionals clearer.

The mechanics for this refactoring are described in [Fow99] as follows:

- Find the value of the control flag that gets you out of the logic statement,
- replace assignments of the break-out value with a **break** or **continue** statement,
- compile and test after each replacement.

```
while (!done && g) {
  if (cond) {
    P

    done = true;
  }

  Q
}
```

```
while (g) {
  if (cond) {
    P
    Q
    break;
  }

  Q
}
```

Figure 6.9: Remove Control Flag Refactoring. The additional occurrence of Q is not described in literature, but is required to ensure equivalence.

Figure 6.9 shows a schema of the refactoring (we only consider the addition of a **break** statement). Observe the additional occurrence of Q in the transformed program. This is not described in the mechanics; however, it is *necessary* to ensure equivalence of the original and transformed program. The following example of a linear search method (a classical application of *Remove Control Flag*) demonstrates this catch.

```
int i = 0;
while (!done && i < arr.length) {
  if (arr[i] == needle) {
    done = true;
  }
  i++;
}

return i-1;
```

(a) Before

```
int i = 0;
while (i < arr.length) {
  if (arr[i] == needle) {
    break;
  }
  i++;
}

return i-1;
```

(b) After

Figure 6.10: Incorrect Application of Remove Control Flag

Example 6.2 (Wrong Application of *Remove Control Flag*)**.** The program in Fig. 6.10a searches an array `arr` for an element `needle`, using a control flag `done` to signal that the element was found. If `needle` exists in the array, it will be at position `i-1` after the loop; otherwise, `i-1` will be -1 (if `arr` is empty) or point to its last element. An application of *Remove Control Flag* simply following the mechanics described in [Fow99] results in Fig. 6.10b, where `done` has been removed (it is not accessed outside the loop) and the assignment to it has been replaced by a **`break`** statement. However, the previous contract is violated: After the loop, `i` (and not `i-1`) points to `needle` in `arr` if it is present. If `needle` cannot be found, value of `i` is the same as for the original program. ◊

The actual model we implemented in REFINITY (complete code in Appendix E, Figs. E.13 to E.15, Pages 355 to 357) is slightly more abstract, as the assignment to `done` and the **`break`** may occur somewhere inside the ASs *P* / *Q*, and not necessarily as a final statement in the **`if`**. We therefore use ASs with *different identifiers* in the transformed program: AS *Q* in the original program is specified to always complete normally, while the corresponding AS, which occurs as last statement in the **`if`**, is specified to always complete *abruptly* due to a **`break`**. Thus, they cannot represent the *same* programs. To anyway represent programs that, albeit completing for different reasons, have the same *effects on the state*, we use *postconditions*.

The frame and footprint of the whole loop are, together, overapproximated by an abstract location set `loopLocs`. The abstract invariant `loopInv(`**`\value`**`(loopLocs), i)` is established by AS *Q*. The value of loop guard *g* is coupled to an abstract predicate `guardVal(\value(loopLocs))`; for *cond*, we use a predicate

```
doneCondition(\value(loopLocs), i).
```

We use a ghost variable `i` counting loop iterations to access the "done" predicate of the *preceding* loop iteration.[6] The whole invariant for the original program is

```
/*@ loop_invariant
  @   (done <==> doneCondition(\value(loopLocs))) &&
  @   loopInv(\value(loopLocs));
  @*/
```

The following specification, which can be added as an AE constraint or as relational precondition in REFINITY, only permits *strongest* invariants as instantiations of the abstract invariant:

```
(\exists any _frL; (\exists boolean _I; (\exists boolean _Done; (
  (\forall any frL; (\forall boolean I; (\forall boolean Done; (
        ((Done <==> doneCondition(frL, I)) &&
         loopInv(frL) &&
         !(!done && guardVal(frL)))
    <==> (_frL == frL && _I == I && _Done == Done)
))))))))
```

This specification is logically equivalent to

```
(\exists any _frL; (\exists boolean _I; (
  (\forall any frL; (\forall boolean I; (
        (loopInv(frL) &&
         !(!doneCondition(frL, I) && guardVal(frL)))
    <==> (_frL == frL && _I == I)
))))))
```

after "inlining" the equivalence for `Done`. Since `Done` no longer occurs, this can be instantiated and used for the transformed program—provided that the abstract invariant is assured. This demonstrates that in relational proofs of abstract programs, it does not suffice

[6] We need this because KeY misses a `\before` directive for abstract locations. For program variables, this has been added in [Lan18].

to use abstract strongest invariants when specifying loops with abruptly completing bodies: Q would then not have to establish the invariant before the **break**. Consequently, proving equivalence would not be possible. We *have to use super invariants* to prove equivalence. The only situation where Q would not have to be pushed inside the conditional is when P alone can establish the loop invariant. Because we use *strongest* invariants, however, the only way for Q to *not* contribute would be to have *no externally observable effects*.

We conclude that *Remove Control Flag*, as in [Fow99], is *unsafe*, and it is easy to break working code by applying it. The informal description suggests pretty simple mechanics: Replace the assignment to done by a **break**, remove done. In fact, the associated shortcut in general breaks equivalence. It is only admissible if equivalence is not required, e.g., because the effects of Q are confined to state which is only internally relevant for the loop (such as a loop counter which is not read outside), or if Q has no observable effect at all. Otherwise, Q has to be executed before abruptly leaving the loop.

6.4 Performance and Discussion

All abstract program models discussed in the last section and shown in Appendix E, including those with loops[7], are proven fully automatically using REFINITY and KeY. We measured strategy execution times and proof sizes of these proofs. Execution times are mentioned to help the reader develop a feeling about the perceived complexity of these proofs; they are not meant to be statistically reliable. All proofs were conducted on a laptop with an Intel®Core™i5–5200U CPU working at 2.20GHz, and 32 GB main memory. We round execution times to full seconds.

The results are visualized in Figs. 6.11 and 6.12. We abbreviated "*Consolidate Duplicate Conditional Fragments*" by "C.D.C.F.". The proofs took between 9 (*Extract Method*) and 182 (*Remove Control Flag*) seconds. In average it took KeY 41 seconds to close a proof obligation generated by REFINITY, the median amounts to 21 seconds. The resulting proofs have sizes between 1K and 18K nodes (mean 4,886, median 3,075). The refactorings with loops, *Split Loop* and *Remove Control Flag*, have the most complex proofs. With a little more than three minutes, the latter took about eight times the amount of time needed to perform the most complex proof *without* loops (*Consolidate Duplicate Conditional Fragments* in the "Extract Prefix" variant), for which 23 seconds were needed.

Since the abstract program models for the refactoring techniques basically only consist

[7] The framework presented in [SH19a] could not yet perform fully automatic proofs of models with loops, but either needed direct human interaction or *proof scripts* for coupling loop executions.

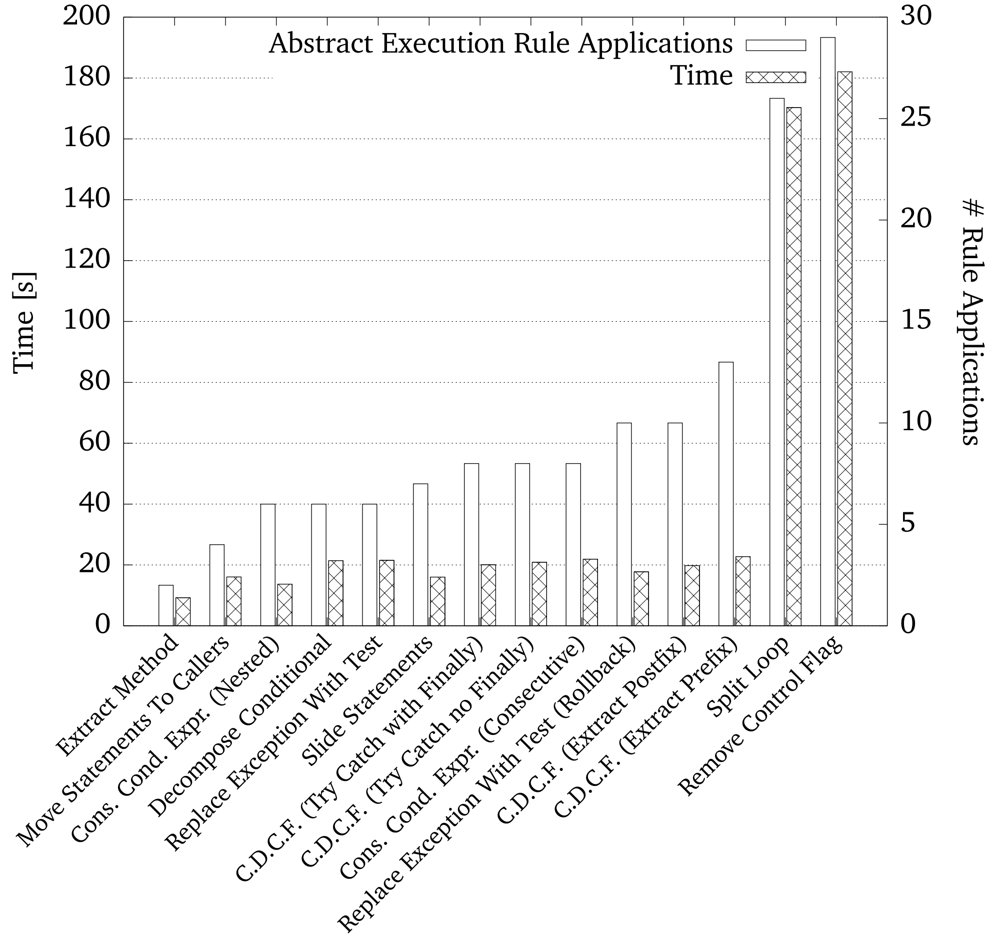


Figure 6.11: Prover Time vs. Number of Abstract Execution Rule Applications

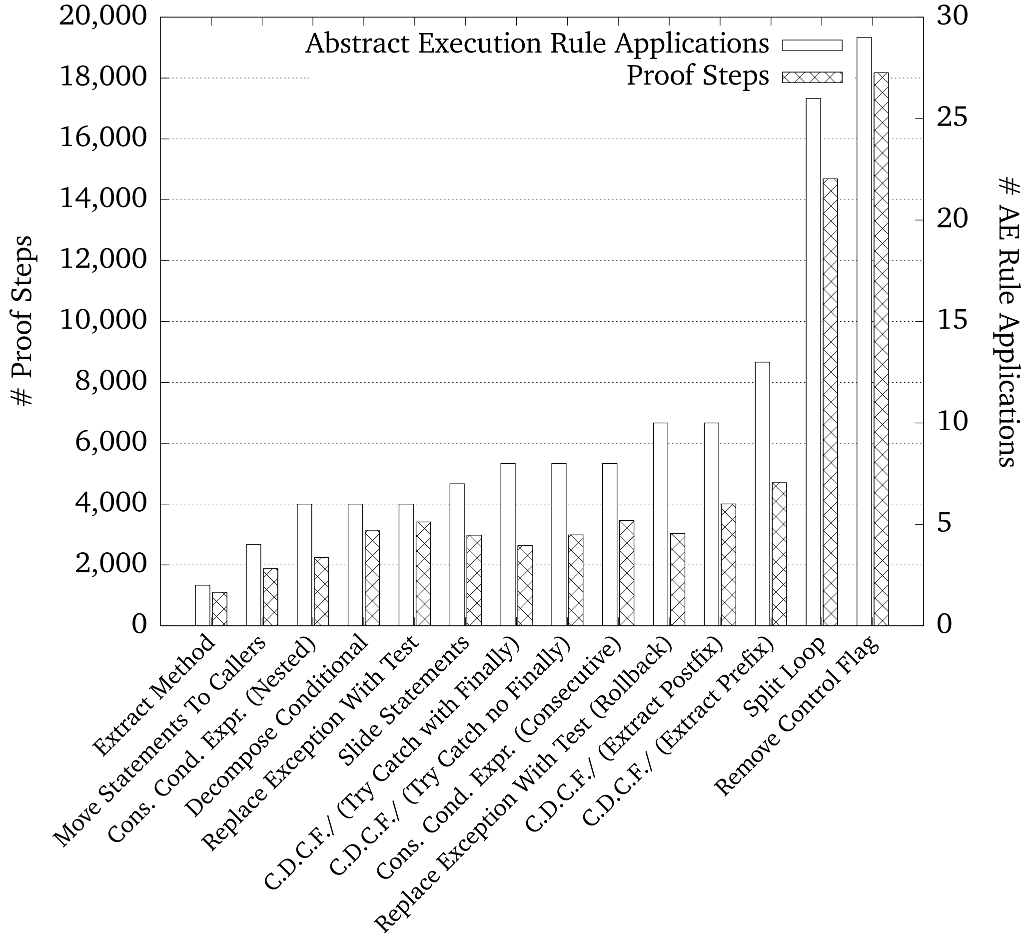


Figure 6.12: Proof Size vs. Number of Abstract Execution Rule Applications

of APEs, control flow instructions and constraints, it is not surprising that the number of AE rule applications is related to the proof sizes, as indicated by Fig. 6.12. Moreover, one can expect that loops complicate symbolic execution, especially when invariants are abstract and bodies may complete abruptly. To shed some light on the influence of APEs and abstract loops on proof sizes and execution times, we conducted two further experiments with artificial abstract programs. In the first one, we created 30 proof obligations containing a single box modality with an uninterpreted predicate as postcondition and one to 30 trivially specified ASs inside the box. Figure 6.13 illustrates the sizes of the resulting proof trees as well as the time needed by the automatic strategies to construct these proofs. The results show that there is a linear dependency between the number of ASs and proof sizes as well as strategy times. The execution of one AS requires a constant number of about 212 proof steps. The measurements for the prover time are more volatile; the median of the contribution of one AS to the proof search was 700 ms.

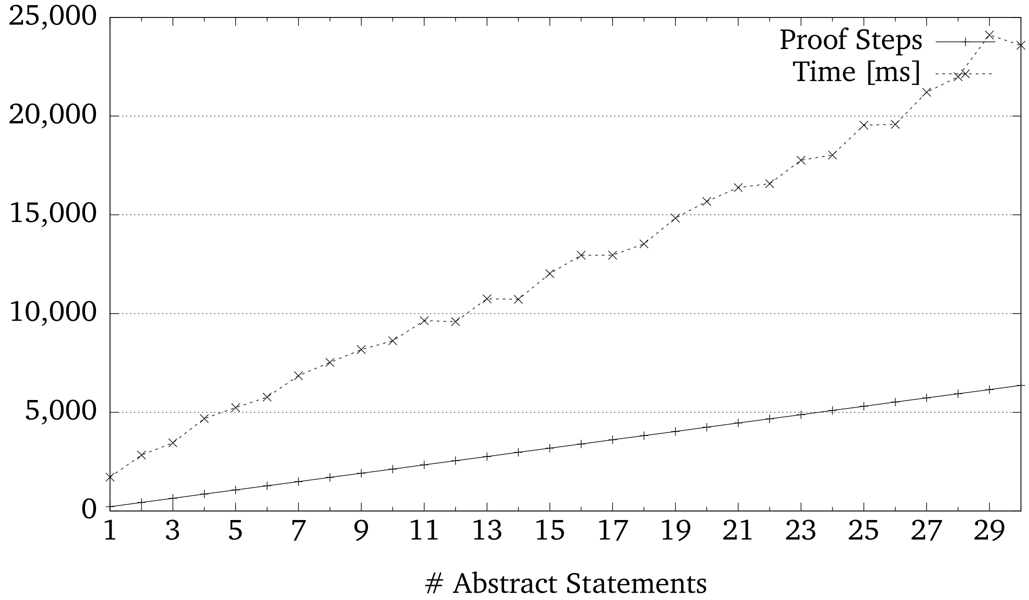


Figure 6.13: Proof Size and Prover Time for AS Sequence Benchmark

In a second experiment, we composed abstract programs consisting of multiple occurrences of the same loop with abstract guard and body. The loop and the contained APEs have a small, but sensible specification of the shape we proposed in Sect. 6.2; Listing 6.5 shows the loop's code. The constructed abstract program has a leading constraint assuming the initial validity of the abstract loop invariant. Apart from that, there are no additional constraints or statements; in particular, the initial validity of a loop invariant for a subsequent loop is assured by the validity of the loop invariant of the loop before. The resulting proofs each have one open goal in which the validity of the uninterpreted postcondition predicate should be shown. A proof for n loop occurrences requires $2n$ applications of AE rules (one for each abstract guard and body). The calculated metrics are displayed in Fig. 6.14. Apart from proof sizes and prover times, we also show the numbers of proof branches and Symbolic Execution steps. Interestingly, we observe a linear increase of proof sizes, branches and SE steps, but a superlinear increase in the time needed for proof search. We assume that this is due to the increasingly complicated expressions arising after many loop invariant rule applications: Each of those adds one *abstract* update anonymizing the abstract loop frame. Since those updates cannot be simplified away without additional preconditions, each "use case" branch has deeper expressions than the previous one, which complicates the matching process for the strategies.

Listing 6.5: Loop of Sequential Abstract Loop Benchmark

```
/*@ loop_invariant loopInv(\value(loopLocs));
  @ assignable loopLocs;
  @*/
while (
    /*@ assignable \nothing;
      @ accessible loopLocs;
      @ normal_behavior ensures (boolean) \result <==>
      @   guardIsTrue(\value(loopLocs));
      @ exceptional_behavior requires false; @*/
    \abstract_expression boolean e
) {
    /*@ assignable loopLocs;
      @ accessible loopLocs;
      @ normal_behavior ensures loopInv(\value(loopLocs));
      @ exceptional_behavior ensures loopInv(\value(loopLocs));
      @ return_behavior ensures loopInv(\value(loopLocs));
      @ break_behavior ensures loopInv(\value(loopLocs));
      @ continue_behavior ensures loopInv(\value(loopLocs)); */
    \abstract_statement P;
}
```

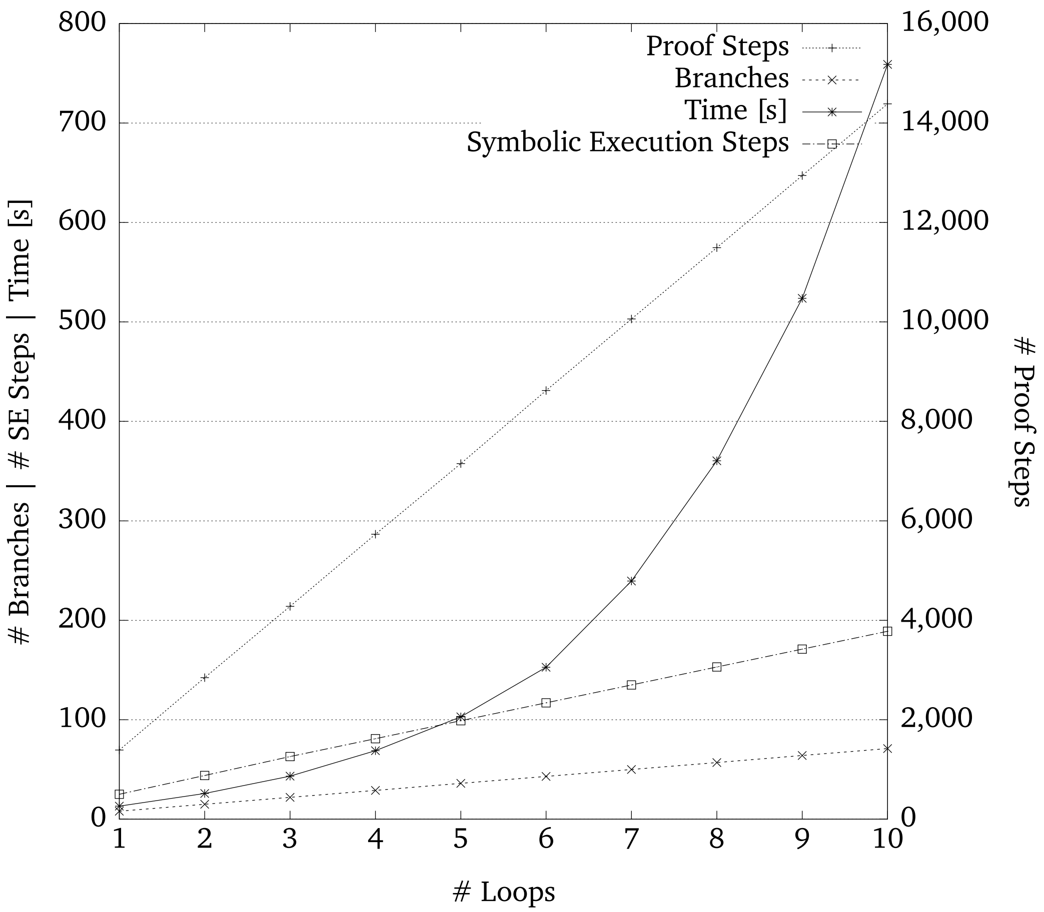


Figure 6.14: Proof Steps, Branches, Prover Times and Symbolic Execution Steps for Sequential Abstract Loops Benchmark

Considering the *syntactic* sizes of our abstract program models, execution times and proofs sizes seem substantial. Reasoning with abstract programs is complicated: Abstract updates cannot be *applied* like their concrete counterparts; instead, the sequent has to be searched for evidence justifying an update simplification. Each APE can have several behaviors. An AS inside a loop can, for instance, complete in five different ways. Each of those can have a complicated pre- and postcondition. Finally, using abstract strongest loop invariants requires existential quantifier instantiation. This did not cause prover incapacities in our experiments, but increases proof sizes and consumes time.

Our recommendation is to try keeping abstract program models as small as possible, and to prefer expressive postconditions over multiple abstract statements. Also then, a proven-correct relational property about abstract programs allows to derive *infinitely many* relational properties about concrete programs—*fully automatically*.

Model Validation The biggest mistake when modeling refactoring techniques is to omit checking the models for sensible instantiations. First, it is always possible that accidentally, an abstract program represents no concrete program at all. This is easiest accomplished by a combination of unsatisfiable constraints. For instance, all models with a leading `//@ ae_constraint false;` are equivalent, but cannot be instantiated. More subtly, the set of instantiations might be non-empty, but not comprise the instantiations it should. A good workflow to develop sensible models is to start from representative concrete programs and to abstract them step by step. This helps to structure the modeling process and to not skip the validation step, since benchmarks already exist. *Formally* validating *by hand* that a concrete program is an instance of an abstract model can be quite tedious, since this involves showing a lengthy proof obligation (cf. Def. 4.5 and Remark 4.7). For the future, we plan to construct a (semi-automatic) instantiation checker also suitable for model validation to assist in this task.

Functional vs. Coupling Loop Invariants AE was designed to be aware of abrupt completion. Likewise, our refactoring models should apply to abruptly completing programs, as long as abrupt completion does not compromise safety of the refactoring technique itself. This leads to the problem that standard coupling techniques (e.g., based on product programs) used in relational verification are not applicable, as discussed in Sect. 6.2. Our remedy, (strongest) abstract strongest loop invariants, arguably increase the specification, and probably also the verification effort: Ideally, one would, at each synchronization point, simply assert "`\result_1==\result_2`". Yet, this approach does not only suffer from complications caused by abrupt completion; it is also difficult to apply to *structurally* *different* programs. Take the *Split Loop* refactoring: It would not have been straightforward to couple one loop in the original to two loops in the transformed program. In contrast, it seems natural that a loop maintaining two separable invariants is split into two loops, each maintaining one invariant. What is more, even comparing recursive with iterative programs is feasible when using strongest invariants / strongest recursive method contracts with the same abstract predicates. Therefore, we think that while it is definitely interesting to look into integrating techniques from relational verification of concrete programs, abstract invariants are conceptually attractive, flexible and practical.

7 Related Work

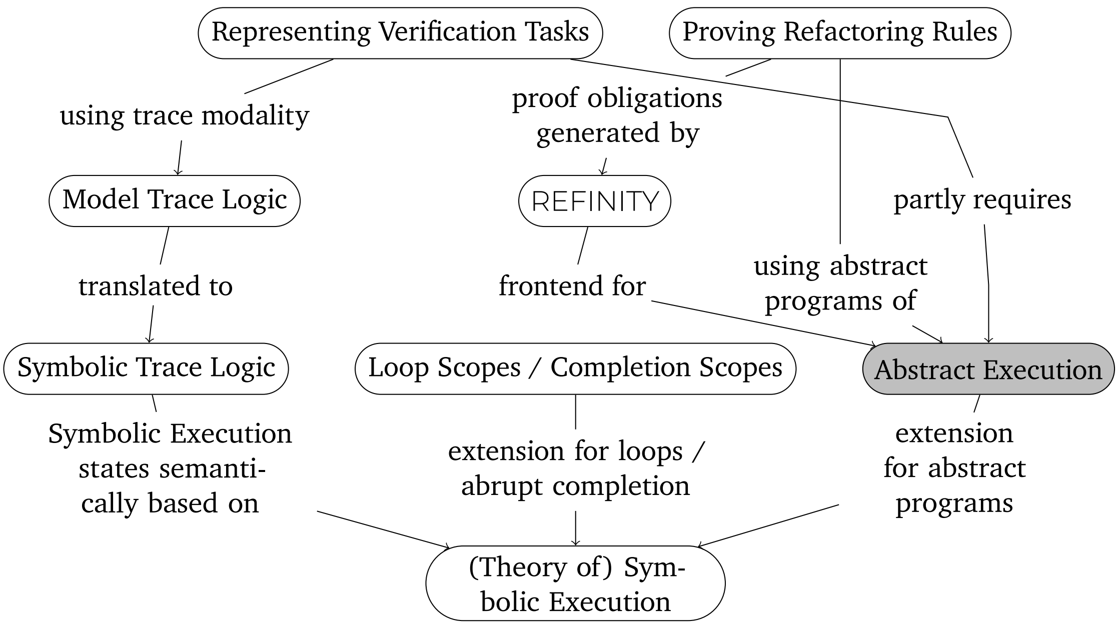


Figure 7.1: Dependencies Between Parts of This Thesis

This thesis intersects with several, partly quite different research areas. Figure 7.1, which we already showed in the introduction, visualizes our contributions and a partial set of their interdependencies.[1] We compare to related results in the following fields:

- Verification of abstract programs (*AE*)

[1] The author of this thesis contributed to, but did not invent (alone), the concepts of loop scopes and completion scopes. Neither, of course, is Symbolic Execution itself a contribution of this thesis. The *theoretic framework* for SE presented in this thesis is an original contribution.

- Validation of refactoring techniques (*Proving Refactoring Rules*)
- (Tools for) relational program verification (REFINITY)
- Approaches unifying program verification techniques (*Representing Verif. Tasks*, *MTL*)
- Logics based on (symbolic) traces (*MTL*, *STL*)
- Foundations of SE (*(Theory of) Symbolic Execution*)
- Handling abrupt completion in SE (of loops) (*Loop / Completion Scopes*)

Verification of Abstract Programs

Schematic programs are a natural artifact in program transformation systems. A well-known example are *compilers*: Arguments for the correctness of the compilation of a particular type of statement, or also of an optimization rule, will typically contain place-holders for subexpressions. Of great interest is the work on *mechanically verified* compilers. CompCert [Ler09] is a verified compiler from C to PowerPC assemblies (mostly) written in Coq. It covers all compilation phases including optimization. A related system is Jinja [KN06], a formalization in Isabelle/HOL of semantics, virtual machine and compiler of a Java-like language including a mechanized proof that the compiler preserves the semantics. CakeML [Tan+16] is a more recent verified compiler for a functional programming language developed in Isabelle/HOL which invests a lot of effort in modularity. The common denominator of these systems is that correctness properties are formalized in an *interactive* proof assistant (Isabelle [NPW02] or Coq [Coq19]) and consequently *proven interactively* using proof scripts. The properties that can be expressed by AE are more restricted (see Sect. 4.2); In particular, we can only reason about the effects of programs and not their internal structure. However, AE is a *semi*-interactive method (implemented in KeY) and *fully automatic* in many cases: We could automatically prove the correctness of all refactoring techniques modeled in Chapter 6. Also, for many applications, abstracting from the inner structure of schematic programs is perfectly admissible. Subsequently, we look at techniques processing abstract programs *automatically*.

Ahrendt et al. [ARS05] automatically validate program transformation rules of KeY's JavaDL calculus against an *executable* semantics in rewriting logic [MR06] implemented for the Maude system [MW91]. They do not consider schematic statements, but do have a notion of *schematic expressions* (corresponding to our Abstract Expressions). The authors "lift" the executable rewrite theory for concrete Java to a rewrite theory for *schematic Java* on *generic states*. The theory is aware of expressions with side effects, and the complication that the same expression may have different side effects and results when executed in different states (for this reason, our *APEs* and abstract updates have explicit footprints). The rewrite theory for schematic Java represents state-dependent evaluation of unknown expressions by symbolic snapshots. Unknown side effects are represented by lists of symbolic locations and values, which trigger conditional transformations of symbolic memory. There are strong analogies to Abstract Execution: A symbolic snapshot is an update accumulated by previous symbolic execution steps. An abstract update has as left-hand side a list of symbolic locations, and as right-hand side a list of symbolic values. The effect of the application of an abstract update is also conditional: If the target location is in the symbolic location list, the state is transformed according to the value list, and left unchanged otherwise. Another interesting parallel is that conditional values "cannot be further evaluated [...] and remain in memory as they are, which is fine since we just aim at comparing two resulting states" [ARS05]. Also abstract updates can usually not be terminally evaluated, which is why we introduced rules like $\mathsf{reorderUpd}_1$ normalizing the shape of expressions with abstract updates.

A closely related approach by Bubel at al. [BRR08] aims to automatically validate the correctness of *derived* SE rules in JavaDL within the calculus itself. They use parametric Skolem variables to represent abstract statements. These variables cannot be "executed", but are "decomposed": A Skolem variable representing a statement is split into a symbolic statement for the side effects on the states, and an **`if`** cascade modeling abrupt completion. The latter is very similar to what our AE rules do. Decomposition to a symbolic statement, however, only advances the analysis if the modality can be "linearized", i.e., if $\langle \mathit{stmt}_{Sk}\ \omega\rangle\varphi$ can be transformed to $\langle \mathit{stmt}_{Sk}\rangle\langle\omega\rangle\varphi$. This is only possible if there is no leading prefix π. In the application of [BRR08], where only *schematic rules* are analyzed, the context can indeed by removed (because it exists in all premises and in the conclusion), and linearization works. For concrete programs, this is no option in JavaDL, which is one motivation for using the more flexible instrument of abstract updates instead. Since abstract updates furthermore have explicit (albeit potentially abstract) assignable and accessible locations, we were able define practically useful simplification rules permitting us to prove interesting properties in abstract contexts.

In summary, both [ARS05] and [BRR08] contribute *systems facilitating execution of schematic Java code*, anticipating many of our ideas and concepts, though in a different flavor. The main differences to our work are

- [ARS05] only supports schematic expressions, [BRR08] only supports schematic *statements*; we have both.
- Both AExps and ASs may complete abruptly. The rewriting logic used by Ahrendt et

al. lacked several interesting Java features, including abrupt completion (Bubel et al. model abrupt completion).

- A strength of AE is its *expressive specification language*. It facilitates, e.g., defining (constraints on) explicit (abstract) frames and footprints. Thus, we can express that the side effects of one APE are irrelevant for the execution of another one. The target application of [ARS05; BRR08] does not require additional constraints. AE is also applicable for relational verification of complex properties; for instance, we can couple the result of an AExp to the exceptional completion of an AS.
- (Abstract) updates powerfully and concisely represent (abstract) symbolic state changes. They are more suitable for complex analyses and feedback to human users than the constructs used in the rewrite theory for schematic Java, and more flexible then "statement Skolem variables", which require linearization (are not removed from modalities) and do not have parameters for frames and footprints.

Interestingly, AE was originally conceived in a similar context: In [SH18], we used it to prove the correctness of *schematic compilation rules* from Java to LLVM IR. For this application, we also did not need to specify APE occurrences in a fine-grained manner.

Godlin & Strichman [GS13] perform "Regression Verification" of closely related versions of the same program. They transform loops into recursive functions and replace recursive calls with uninterpreted function symbols. The latter are similar to AExps; However, side effects or irregular termination cannot be modeled, because functions are pure. Since the authors prove equivalence of *concrete* programs and only need abstraction to handle recursion, they do not need, and do not support, ASs.

Mechtaev et al. [Mec+18] propose a mechanism for proving existential second-order properties over symbolic functions. Using *program synthesis techniques*, they construct existential witnesses for symbolic functions from a user-specified grammar. The use of symbolic functions is similar to [GS13]. The main difference to [GS13] and our work is that they target *existential* and not universal properties. Again, we also have ASs, while symbolic functions are more similar to AExps.

The PEC system [KTL09] for proving the correctness of compiler optimizations uses *meta variables* ranging over expressions, variables and statements. The latter are "single-entry-single-exit", whereas ASs can have multiple exit points, including abrupt completion. The property to be proven in [KTL09] is a certain bi-simulation relation which is somewhat inflexible and requires lockstep execution. As all other approaches discussed, also this one does not support specification of abstract elements to constrain their instantiations.

Alive [Lop+18] permits proving automatically the correctness of "peephole optimizations" for LLVM. Local algebraic simplifications and code optimizations are expressed in a restricted DSL, and then transformed to first-order logic assertions that are passed to an SMT solver. While this approach reasons about classes of programs, it is parametric only in register names and imposes other serious restrictions (e.g., no loops).

Validation of Refactoring Techniques

We distinguish two lines of research: (1) *Static* formal verification of refactoring techniques, including extraction of preconditions using formal methods and static enforcement of safe refactorings, and (2) *dynamic* validation techniques using testing and runtime assertions.

Starting with (1), Garrido & Meseguer [GM06] formalize the Java refactoring techniques "*Pull Up / Push Down Field*", "*Pull Up / Push Down Method*", and "*Rename Temporary*", based on the executable Maude semantics for Java also used by [ARS05].[2] They prove the correctness of *two* refactorings by a mixture of Maude evaluation and pen-and-paper argumentation, and define the preconditions under which the refactorings can be applied. Using AE, we were able to create *fully mechanized* proofs of *all* refactoring models we created. Since the preconditions we provide are used in these proofs, they are guaranteed to ensure the safety of the refactoring techniques.

Schäfer et al. [Sch+12] address the problem of naming and accessibility in refactoring: Any refactoring introducing, moving, or deleting a declaration may run into problems with the program's binding of names to declarations. Similarly, when moving a reference to a declaration, care has to be taken that this reference is still bound to the same declaration after moving. The proposed approach consists in a translation from Java to a lookup-free and access-control-free representation J_L implemented in the JastAdd compiler framework [HM03]. Refactorings are then expressed on the level of J_L, where transformations cannot accidentally change name bindings or introduce unbound names. This approach is orthogonal to ours: AE is oblivious of concrete names. In the KeY system, which serves as a host for our reference implementation of AE, it is even possible to distinguish *different* program variables with the *same names*. On the other hand, Schäfer et al. do not consider other problems than those related to naming and accessibility. For a safe application of refactoring techniques in practice, our discovered behavioral constraints should indeed be *combined* with a framework aware of names and bindings.

[2] Garrido & Meseguer report that they proved the correctness of *five* refactoring techniques. Since we always prove *equivalence* of original and transformed programs, we would only count them as three, since, e.g., *Pull Up Method* is the inverse of *Push Down Method*. The preconditions we give for *Extract Method*, for instance, can also be used for a safe application of *Inline Method*.

Also the work by Silva et al. [SSM15] addresses non-behavioral properties. The authors use Alloy [Jac06] to verify the type correctness of Java code transformations and the satisfiability of their specifications, and assert that in their approach, “the only potential issues caused by a transformation are behavioral ones”. They use testing to check for deviations in the program behavior, but do not explain this in detail.

Regarding line (2), Soares et al. [Soa+10] use static analysis to automatically generate a test suite for detecting behavioral changes. They evaluated their implementation for a number of (high-level) refactorings and found actual errors in real applications. Eilertsen et al. [EBS16] bring forward the idea of “improved refactorings”, which introduce semantic correctness assertions for the preservation of program behavior when transforming the input program. They support two refactoring techniques, *Extract and Move Method*, and *Extract Local Variable*. The authors regard their approach as a more “fine-grained” attempt compared to Soares et al., since they can inspect the heap structure in more detail compared to observing the output of unit tests. A related approach addressing general program transformations (and not refactoring in particular) is proposed by Namjoshi and Zuck in [NZ13]. They suggest to augment every implemented transformation (e.g., a compiler optimization or a refactoring technique) by a *witness generation* procedure. For every application of the transformation, this generator constructs a relation which guarantees the correctness of the instance.

Our work is, to the best of our knowledge, the only one supporting modeling and automatic verification of behavioral equivalence for statement-level refactoring techniques. When used in practice, it should be complemented with checks for details such as name binding and accessibility (e.g., [Sch+12; SSM15]). On-the-fly formal verification of the conformance of concrete programs with behavioral preconditions for refactoring techniques is, as of now, infeasible. We think that it is an interesting idea to follow the idea to generate assertions, or correctness witnesses, from these preconditions that can be dynamically checked at runtime. Notwithstanding, dynamic approaches cannot, for example, guarantee the absence of (possibly indirect) changes to the heap. This *can be assured* by modular deductive program verification with strong frame conditions. For highly safety-critical applications, formal verification of refactoring preconditions should be considered, and is arguably easier than functional verification of whole units.

(Tools for) Relational Program Verification

The principal use cases for AE reside in the area of *relational verification* [BU18], which includes, but is not limited to: general-purpose relational program proofs [BCK11;

KKU18], correctness proofs for refactorings [GM06; Sch+12; SSM15], regression verification [GS13], proven-correct compilation [Ler09; Tan+16] and compiler optimizations [Lop+18; KTL09], program synthesis [SGF10], and information flow properties (e.g., by self-composition [BDR04; DHS05]). We have discussed some of these above. Here, we focus on *conceptual aspects* and *tool support* for general-purpose relational verification.

Barthe et al. [BCK11] propose the construction of *product programs* from two variable-disjoint programs as a general-purpose technique for verifying relational program properties. After execution of the product program, the result is checked for correctness, e.g., equality. This works also for structurally different programs, although it might not be possible to construct the product *automatically*.[3] The main advantage of product programs is the *reduction of relational to functional verification*, facilitating usage of already existing tools for the functional case. In REFINITY, programs are executed in isolation. This spares the pre-processing steps of renaming of locations in one program and constructing the product of the input programs. The latter, done wrongly, can invalidate the whole analysis. The programming language used in [BCK11] does not know abrupt completion, which significantly complicates product construction. On the other hand, we need *strongest functional* loop invariants, which are usually difficult to find for concrete programs. Product programs and AE are not mutually exclusive: One can create a *product* of *abstract* programs. In a later extension [BCK13], Barthe et al. propose a more general framework for *asymmetric* relational problems, where traces may be universally or existentially quantified. We discuss this work in the next section.

Kiefer et al. developed LLRêve [KKU18], a tool for automatically verifying the equivalence of C programs. Input programs are translated into LLVM IR. The control flow graph of the program in the intermediate representation is then divided into linear segments. For points at which these segments are connected as well as for pairs of corresponding function calls, relational abstractions using uninterpreted predicate symbols are introduced. Finally, constraints (in Horn normal form) over the predicate symbols linking the linear segments are generated, which are passed to a constraint solver for Horn clauses. LLRêve is an example for a system highly specialized for relational program verification, including a fine-tuned calculus for that purpose. The focus is on automation: For sufficiently similar programs, synchronization points are inferred automatically. Otherwise, the user can help the prover by specifying them manually. To better support handling structurally different programs, dynamically collected information is used to harmonize the loop structure and infer invariant candidates. The tool has a web frontend[4] which allows specifying two programs, choose a solver and start the automatic analysis. Compared to REFINITY,

[3] "Building a product program from a pair of components is undecidable in general". [BCK11]

[4] `https://formal.iti.kit.edu/projects/improve/reve/`

LLRêve, or at least its web frontend, only supports equality, and not arbitrary relations, as postcondition. In addition, we did not find any special support LLRêve provides for abrupt completion. However, REFINITY does not know coupling predicates and requires functional contracts. While this is acceptable for abstract programs, it constitutes a shortcoming for applications to concrete programs. We think it would be interesting—and challenging—to apply the LLRêve approach to REFINITY/abstract programs. For example, we could dissect Symbolic Execution Trees created for two abstract programs into linear segments which are then coupled by abstract predicates.

SymDiff [Lah+12] by Lahiri et al. is a "differential program verifier" operating on the intermediate verification language Boogie. Loops and unstructured control flow are translated to tail-recursive procedures. The tool uses *mutual summaries* [Haw+13] which are coupling predicates relating summaries of two (possibly recursive) programs, thus generalizing postconditions used for single program verification. The summaries are not automatically inferred; loop optimizations (such as unrolling) have to be triggered by the user. Integrating mutual summaries into REFINITY is also an interesting option, and would not imply replacing the whole backend, which we would have to when incorporating the LLRêve approach. Since we are working on the level of Java source code, we would have to find our own way to encode control flow attributed to abrupt completion, in particular of APEs, into tail recursion. Moreover, such an encoding would dilute the connection between the proof tree and the model, making it more difficult to draw conclusions from failed proof attempts. As the verification of abstract programs is in principle more advanced than verifying concrete programs, we think that this connection is even more important for REFINITY than for concrete program verifiers.

Further related work on relational program verification focuses, for instance, on comparing programs with different iteration structures [Ber11; VJB12] or more liberal coupling predicates [BNN16]. For a more detailed overview, we refer to, e.g., [BU18]. None of the discussed approaches can handle abstract programs; they generally outperform REFINITY when it comes to *automatic* relational verification of *concrete* programs. A distinguishing feature of REFINITY as a *frontend* for relational verification is our approach to specify arbitrary relational postconditions: A user selects for each program the locations of interest, and uses them in a JML formula to define the relation to prove. As a default, REFINITY specifies full equality. Furthermore, as we build on top of the semi-interactive program prover KeY, the user can inspect, and interact with, the resulting proof tree if needed.

Approaches Unifying Program Verification Techniques

Building on their concept of product programs [BCK11], Barthe et al. propose *asymmetric* product programs [BCK13]. Those are based on a control-flow graph representation rather than on concrete syntax, allowing to relate programs in different languages. Central to the approach is the notion of a *left product*, which expresses: For every execution of the first program, there is a related execution of the second program. In other words, the second program *approximates* the first program. This allows to integrate non-deterministic statements and to compare programs with different termination behaviors. Asymmetric product programs are closely related to TDL's notion of the trace modality: Barthe et al. allow to compare programs in different languages, *provided they support a control flow graph representation*; the trace modality integrates different languages, *as long as they support a trace representation*. The trace modality also expresses that its right side approximates its left side. More specifically, a product of P_1 and P_2 is a left product if it can progress at any point where P_1 can progress, while the semantics of $[P_1 \Vdash P_2]$ is the set of all traces that, if they are a trace of P_1, are also a trace of P_2. Program P_2 consequently approximates P_1 if all traces of P_1 are contained in the traces of the left product $P_1 \ltimes P_2$ / of the trace modality formula $[P_1 \Vdash P_2]$. To allow for more relaxed properties than full trace inclusion, Barthe et al. use *partial specifications*, which are generalized relational pre- and postconditions. We use *trace abstractions* instead.

The main theorem about asymmetric products in [BCK13] relates judgments about "refinement quadruples" $\{\varphi\}\, P_1 \mapsto P_2 \,\{\psi\}$ to left products. Informally speaking, a refinement quadruple is valid if either P_2 does not terminate, or if for each trace of P_1, there is a corresponding trace of P_2 such that the initial states satisfy the relational precondition φ and the final states satisfy the relational postcondition ψ. This is true if there is a left-product preconditioned with φ which is correct w.r.t. ψ. To state this in TDL, we introduce the *postcondition abstraction* α_ψ. Let ψ be a relational postcondition for variable-disjoint programs P_1 and P_2. We lift ψ to a property Ψ of pairs of states. For instance, if $\psi \equiv \mathsf{x} \geq \mathsf{x}'$, Ψ contains all pairs (σ, σ') where $\sigma(\mathsf{x}) \geq \sigma'(\mathsf{x}')$. Then, $\alpha_\psi(\mathcal{T}) := \mathcal{T} \cup \{\tau' \in \mathit{Traces} \mid \exists \tau \in \mathcal{T};\ \Psi(\mathit{last}(\tau), \mathit{last}(\tau'))\}$. We now can reduce $\models \{\varphi\}\, P_1 \mapsto P_2 \,\{\psi\}$ to $\Vert\!\models \varphi^{ltl} \land \langle P_2 \Vdash \mathrm{true}\rangle \rightarrow [P_1 \Vdash_{\alpha_\psi} P_2]$. Note that α_ψ entails big-step abstraction (of finite traces). We need the premise $\langle P_2 \Vdash \mathrm{true}\rangle$ because the refinement quadruple is always valid whenever P_2 does not terminate. This example underlines the versatility of the trace modality and trace abstractions, and, in our opinion, makes the meaning of refinement quadruples more explicit. We would like to point out that in [BCK13], the validity of a judgment about refinement quadruples is *reduced to* a statement about left products, which does not mean that this is the only judgment that can be evaluated with this instrument. It would be interesting to attempt a characterization of properties provable with left products in the framework provided by MTL.

Kamburjan [Kam19] proposes Behavioral Program Logic (BPL), a dynamic logic for trace properties integrating behavioral types and allowing for reasoning about non-functional properties within a sequent calculus. BPL has the *behavioral modality* which asserts that a statement in a concurrent language meets a behavioral specification consisting of a *type* and a *translation* of the type into an MSO formula. This is the case if that formula holds for all traces generated by the statement. Important differences to our approach include: (a) BPL *syntactically* integrates analyses on the *same program class*, while MTL is a *general semantic framework*, (b) the translation of [Kam19] projects to MSO, and is thus less expressive than MTL, which projects atoms to arbitrary trace sets. This reflects our aim to be as general as possible, while [Kam19] has a specific target application (combining certain analyses for an active objects language). The trace modality can also be used to combine verification techniques: Two specifications can be combined by forming the intersection of the trace sets. In contrast to MTL, BPL has a calculus, which is possible since a significantly larger part of its syntax is fixed. To reason about MTL statements, one can translate to the symbolic traces of STL and use its sequent calculus, which is incomplete, yet very abstract and thus quite general.

Some systems do not provide a common semantics for verification domains, but a framework to *implement* different analyses. They usually represent verification problems in an Intermediate Language (IL) and interface to different provers. Boogie [Bar+05] and Why3 [Bob+11] both are an IL and tool for deductive program verification. They are used as backends by verifiers for languages like C and Java. When translating MTL problems to STL, STL's regular symbolic trace language can be regarded as our “IL”. Symbolic traces are, compared to Boogie and WhyML, less usable for direct programming, more abstract and less expressive (e.g., we cannot directly write loops, but have to use invariants). Yet, the syntactic notion of symbolic traces is closely related to the semantics of MTL (sets of traces), allowing *formalizing* and *proving* a problem in a closely related framework. Moreover, MTL/STL can readily express other problems than “standard” postcondition verification. The STL calculus also interfaces to different provers: Which one to use for symbolic state subsumption checking is left open.

Logics Based on (Symbolic) Traces

De Giacomo & Vardi [DV13] propose a Regular Temporal Specification language RE_f that is syntactically similar to STL symbolic traces, but ranges over *propositional* formulas while our atoms are first-order symbolic states. They show that RE_f has the same expressiveness as MSO and is strictly more expressive than LTL on finite traces. They define Linear-time Dynamic Logic LDL_f, having the same expressivity as RE_f, but allowing logical connectives like negation. The reasoning system for LDL_f is based on a translation to automata. De Giacomo & Vardi mention, but do not detail, the possibility to "capture finite executions of programs [...] (in a propositional variant [...])". We do this—only not restricted to a propositional variant or *finite* executions. In addition, we incorporate *abstract programs* to reason about *classes* of programs. It would be interesting to investigate whether we could use a variant of LDL_f to embed symbolic traces conveniently into logic formulas.

Barthe et al. [Bar+19] present a *trace logic* for verifying relational properties. It explicitly reflects time points and program locations, and has an explicit syntactic category of trace symbols. Conceptually, programs are translated to sets of axioms describing their behavior; then, partial correctness w.r.t. a relational postcondition is expressed as the entailment of the postcondition by the axioms. The approach is implemented in the Rapid tool[5] and uses the first-order theorem prover Vampire for proving validity. Trace logic can be regarded as an expressive TDL in the context of MTL, which could be used to formalize specifications in trace modality formulas. More importantly, since programs as well as formulas can be translated to trace logic, it could replace STL as a backend for verifying MTL problems. This idea is attractive, as it allows to use the strength of existing first-order solvers. The main technical problem to be solved is probably the application of trace abstractions to trace logic formulas; Possibly, the translation to trace logic already has to be abstraction-aware. This can be seen as an advantage of symbolic traces, which allow for a decoupling of the translation and abstraction steps (similarly to the semantics of the trace modality). MTL is better suitable than trace logic for *communicating* (as opposed to proving) verification problems, due to its much higher abstraction level.

Beckert & Bruns [BB13] combine dynamic logic and first-order temporal logic to a *Dynamic Trace Logic*. They have a trace-based semantics for a while language and provide a sequent calculus to reason about temporal properties (not preceded by a translation to symbolic traces). The calculus rules depend on the top-level operator of the first-order LTL postcondition. This leads to quite complex loop invariant rules. Also, the approach is not directly applicable to other verification domains, e.g., relational verification. Our combination of MTL/STL is more flexible: Left and right-hand side of the trace modality are interpreted in isolation, and STL is completely oblivious of the original Trace Description Language from which symbolic traces were generated.

Din et al. [Din+17] propose a trace semantics for the Active Objects language ABS.

[5] `https://github.com/gleiss/rapid`

Traces are "locally abstract, globally concrete": at the local (e.g., method) level, *symbolic* traces are used. These are primarily a *semantic* notion, facilitating a modular semantics for a concurrent language, while our symbolic traces are *syntactic* entities. The authors briefly sketch a program logic with trace formulas, but leave the notion of trace formula abstract.

Foundations of SE

There is plenty of work on practical aspects of Symbolic Execution, e.g., concerning *dynamically allocated memory* [Xie+05; DLR06; BDP15], addressing the *path explosion problem* [CS13; Bal+18; Yan+19] by subsumption [APV06; Jaf+12; JMN13; CJM14], method contracts [Ahr+16], value summaries [Sen+15] or state merging [SHB16], or increasing its practicability by combining concrete and symbolic execution into *"concolic" execution* [GKS05; Cad+06; HT08]. We do not discuss these aspects here, and instead focus on work on the *formal foundations* of SE. Apart from our contributions, we know of three further approaches pursuing the establishment of such a foundation, thus contributing to a better understanding and justification of the technique itself.

The earliest work we found is by Kneuper [Kne91] from the early '90s. The author defines the "denotational semantics of symbolic execution of specifications and programs", aiming to thus provide a general correctness notion. Two versions of an SE semantics are defined: *Full* SE which precisely captures the set of all possible execution paths, and *weak* SE which overapproximates this set. The latter is frequently required in programs with loops or recursive methods, where invariants or method summaries are needed as auxiliary specifications. Compared to [Kne91], whose definitions of full and weak SE constitute important preparatory work, our framework is more general, since it permits state merging, and more concise, as with JavaDL, we dispose of a richer formal basis.

Lucanu et al. [LRA17] propose a language-independent theory of SE based on a definition of a language's semantics by *term rewriting*. The framework can be extended to a deductive system for proving programs w.r.t. *Reachability Logic* properties and is implemented based on the $\mathbb{K}$ system [RS10]. The authors define two desirable properties of SE and prove that they are satisfied in their framework: (1) *Coverage*, i.e., that to every concrete execution there corresponds a feasible symbolic one, and (2) *Precision*, i.e., that to every feasible symbolic execution there corresponds a concrete one. In contrast to our work, there is no notion of *symbolic state*; SE and its properties are defined in terms of transition systems and their relation to concrete transitions. Instead, we start with the meaning of symbolic states defined by their *concretizations* projecting to concrete sets of states, and then regard properties of transitions based on their effects to these sets. We do not need to relate to a concrete transition system since we include program counters in symbolic states and consider them in concretizations. Thus, our framework also allows more abstract steps: The symbolic transition system does not have to *simulate* the concrete one (or vice versa), only the result of symbolic execution has to satisfy properties that can be related to concrete transition systems via the programming language semantics (big-step interpretation). Our system is language-independent, too; we use an underspecified language interpreter ϱ. It is unclear whether Lucanu et al. also support m-to-n transitions between configurations, i.e., flexible state merging, like our framework.

The most recent formal definition of SE (apart from ours) is by de Boer and Bonsangue [BB19]. They have symbolic "configurations" which correspond to our symbolic states.[6] These are also triples of path condition, store, and program counter, only in reversed order. Similarly to [LRA17], de Boer and Bonsangue abstain from defining the semantics of symbolic states, and directly relate a symbolic transition system to one for concrete execution via simulations, aiming to establish two correctness properties. Those properties are the same ones as in [LRA17] (which are explicitly referred to), namely coverage and precision. The authors of [BB19] prefer to use the notions of *correctness* for coverage and *completeness* for precision. The major difference to Lucanu et al. is that the formalization is not embedded in a comprehensive logic framework, i.e., it is easier to comprehend, but also does not profit from properties that come "for free" with the logic. The differences to our work comprise those already discussed for Lucanu et al. Additionally, de Boer and Bonsangue examine the behavior of their system for languages with recursion, object orientation and multithreading, which was not in our scope since we only aimed at representing fundamental semantic aspects of SE.

We think that our framework is the closest one to providing a general, *semantic* foundation for Symbolic Execution, inasmuch as it starts from the semantics of the basic building block, the *symbolic state*, and builds general m-to-n transition relations and correctness notions on top of that. Since we do not require simulation relations, but rather relations on the represented state spaces by input and output symbolic states, our framework is more flexible. On the other hand, it does not have a notion of "progress" which comes implicitly when using simulations; We could define this as an addition to our system.

Concluding this discussion, we relate the different properties defined for SE in the papers discussed above. *Full coverage / Full precision* by Lucanu et al. corresponds to *correctness / completeness* by de Boer and Bonsangue. Precision as defined by Lucanu et al. *implies* our notion of *precision* (as we do not require simulation), while coverage implies

[6] We also use the term *configuration*, but for a *set* of symbolic states representing the current status of symbolic evaluation; or, in "deductive verification wording", the "open goals" of SE.

our notion of *exhaustiveness*. Kneuper's *full* SE is equivalent to a system with full coverage *and* precision, while *weak* SE requires coverage only. For our system, we deliberately decided against using notions from program proving (correctness, completeness) since this implicitly restricts the application area. While a "correct" system used for program proving has to be *exhaustive* / have *full coverage*, a "correct" system used for a bug finding system trying to avoid false positives mainly has to be *precise*. We prefer "exhaustiveness" over "coverage" because it naturally can be turned into an adjective.

Handling Abrupt Completion in SE (of Loops)

We use loop scopes for graceful and efficient handling of abrupt completion of loops in heavyweight Symbolic Execution; completion scopes (**`exec`** statements) are a, yet unimplemented, generalization. It is natural to compare our approach to control abrupt completion with other heavyweight SE systems like VeriFast[7] and KIV[8]. For VeriFast, an SE system for C and Java, we unfortunately could not find any work formally explaining the handling of irregular control flow in loops; the most formal paper we encountered [VJP15] is based on a reduced language without **`throw`**s, **`break`**s and **`continue`**s. KIV is a deductive verification system which has been extended by an SE calculus covering Java Card in a PhD thesis by Stenzel [Ste05]. Their calculus is also a variant of Dynamic Logic. Its most significant difference to JavaDL is the *flattening* (sequential decomposition) of statements. This implies that the system cannot use inactive prefixes, but instead includes *mode information* in a store shared by multiple modalities, and multiple artificial statements dealing with method returns and abrupt termination. Interestingly, their loop invariant rule bears a strong resemblance to the loop scope invariant rule. Where we decide whether to prove the invariant or the "use case" based on the loop scope index, they decide based on the evaluation of the loop guard and on the mode information. But there are some relevant aspects which distinguish this work from ours: (1) The rule in KIV requires substantially more program transformation due to the flattening. Moreover, we can directly treat **`continue`** statements, whereas they are transformed to labeled **`break`**s in KIV. One of their arguments is that **`continue`**s are problematic for loop unwinding; however, as discussed in [Was16], loop scopes can also be employed in that context, making the transformation superfluous. (2) In [Ste05], the rule circumvents the need for anonymization by dropping the preconditions Γ, which makes it necessary to also encode information about the initial state in the invariant, thus bloating it more than

[7] `https://github.com/verifast/verifast`
[8] `https://www.uni-augsburg.de/de/fakultaet/fai/isse/software/kiv/`

necessary. (3) After abrupt completion, KIV has to process all subsequent modalities until an appropriate "catcher" statement appears. Our approach simply exits the loop scope, which emphasizes the advantages of the "sandboxing" technique. (4) Our work is, to the best of our knowledge, the only one comparing the performance of a "classic" invariant rule to one of this style, and the only one integrating an invariant rule with symbolic state merging. Current versions of KIV can no longer parse Java programs, hence it was not possible to practically examine the implemented rule.

Much work on the verification of sequential programs is based on Verification Condition Generation (VCG). ESC/Java(2) [FS01; Fla+02; Lei05] and its successor OpenJML [Cok14] generate verification conditions for annotated Java programs. The Frama-C plugins Jessie and Krakatoa [MPU04] translate annotated C and Java programs into the Why [Bob+11] language. Boogie [BL05; Bar+05] generates verification conditions for Spec#. In these approaches, the verification works via a translation to an intermediate language. The way loops are commonly translated ("loop framing", [Bar+05]) is structurally similar to our approach: The invariant is asserted initially, accessed locations are anonymized and the invariant is assumed for the anonymized state; finally, the invariant is asserted after executing the loop body. The handling of abnormal control flow depends on the translation into the intermediate language. In OpenJML, statements causing abrupt completion are translated to **`goto`**s to special blocks.[9] Generally, verification conditions consist of one huge implication per method, including one conjunct for each program block ending in a **`goto`** [BL05]. A strength of our approach is the "sandboxing" of the loop body within the loop scope, which makes sure that the loop can only be exited in a very clear and easy-to-comprehend way. This facilitates a modular analysis of loops, since we do not have to deal with the potentially very complex control flow in the CFG induced by an unstructured programming language. Additionally, we require very little program transformation. The translation into an intermediate language may mitigate language complexity; however, it can require compromises concerning soundness [FS01] and, in any case, is a non-trivial and error-prone task [MPU04] which is difficult to prove sound.

Huisman and Jacobs [HJ00] extend Hoare logic for Java-like languages for reasoning about abnormal control flow. They formalize the Java semantics in type theory with special constructs for explicitly catching **`break`**s, **`continue`**s etc., which transform the "abnormal" states back to normal. This idea of "catching" abrupt completion beyond exceptions is closely related to *completion scopes*. In the translation of loop statements, the loops are wrapped into the construct for catching **`break`**s, which resembles our loop scope approach. On the other hand, their framework is based on separate "correctness

[9] Source: Personal communication with David R. Cok.

notions" for all the cases of abrupt loop termination, which is closer to the invariant rule of [Ahr+16]. In our approach, the decision about which property to prove after loop termination is handled in a more "natural" way: By very simple rules that are applied at positions in the proof where the reason for the loop termination can be easily identified.

8 Future Work

> The mind is furnished with ideas by experience alone.
>
> *John Locke*

During our work on the various techniques and applications presented in this thesis, many ideas arose on how to build on it to increase—or demonstrate—their quality. We outline some of these ideas in this chapter, starting with Abstract Execution and its applications, continuing with our trace-based framework of Modal Trace Logic and Symbolic Trace Logic, and concluding with foundational and practical aspects of Symbolic Execution.

Abstract Execution

Mode-Dependent Frames Our specification language for Abstract Program Elements facilitates precise representation of the set of legal instantiations by constraining what locations may be read and written, under which occurrences the APE completes abruptly, and what properties have to hold for the resulting state (the latter is expressed as functional postconditions). Still, there is room for increasing the expressivity of our specifications. We already mentioned in Chapter 4 the idea of *mode-dependent frames*: Specifications of different frames and footprints depending on the completion mode. That is, instantiations of an APE could, for instance in the case of a thrown exception, assign a different set of locations than in the case of normal completion. This behavior can, to some degree, already be realized using postconditions, as demonstrated in Listing 8.1: AS `P` may assign two disjoint dynamic frames `nFrame`, `excFrame`. We store their previous values in ghost variables, and ensure that for normal (exceptional) completion, the previous value of `excFrame` (`nFrame`) is retained. This solution once again illustrates the versatility of our specification language. Yet, it is quite difficult to process such a specification in an automatic proof; Different abstract updates created for different mode-dependent frames could make better use of the existing simplification rules. Specifying different *footprints* for different completion modes, which could be interesting for certain applications, e.g., in the area of information flow security, is much harder to realize using pre- and postconditions (one could choose a formalization similar to the footprint-related part in the definition of the semantics of APEs in Sect. 4.2).

Listing 8.1: Simulation of Mode-Dependent Framing Using Postconditions

```
//@ ae_constraint \disjoint(nFrame, excFrame);
//@ ghost any oldExcFrame = \value(excFrame);
//@ ghost any oldNFrame = \value(nFrame);

/*@ assignable nFrame, excFrame;
  @ normal_behavior ensures \value(excFrame) == oldExcFrame;
  @ exceptional_behavior ensures \value(nFrame) == oldNFrame;
\abstract_statement P;
```

Instantiation Checking In Chapter 6, we explained how we used AE to elicit non-trivial preconditions for statement-based refactoring techniques. This already is useful: Apart from contributing to the understanding of refactorings, we can, for instance, automatically create assertions when refactoring code to increase our confidence in the correct behavior of the transformed program. However, there is so far no *automatic* procedure to *verify* (as opposed to "test") that a concrete program fragment is indeed a legal instance of an abstract one. This is especially interesting for applications of AE such as Correctness-by-Construction (CbC) (see "Applications of Abstract Execution"). We plan to implement an instantiation checker as one of our next steps.

Evaluation and Increased Support for Heap Properties Because we based AE on the theory of dynamic frames, *instantiations* of abstract program fragments already comprise programs manipulating the heap in many ways. Until now, though, our case studies do not cover examples with non-trivial *concrete* heap operations, e.g., with APEs assigning segments of arrays or elements of complex object graphs. As a part of two ongoing or planned research collaborations (in the areas of CbC and loop parallelization), we project to examine the behavior of the existing framework in presence of such operations, and to improve heap support when indicated.

Prototyping in Different Host Framework We conducted a pen-and-paper proof for the soundness of our Symbolic Execution rules for ASs and AExps. To further increase the confidence in the technique, we plan to implement AE as part of an already existing, prototypic formalization of SE for a WHILE language with abrupt completion in Coq we implemented earlier (cf. section "Symbolic Execution" further below). We aim for a *mechanized* proof of the soundness of the principles of AE (or "exhaustiveness" in the terminology established in Chapter 3) based on that implementation.

The core of the AE calculus are *abstract updates*. We think that our previous work with KeY, of which (concrete) updates are an important constituent, did not only strongly influence the present *shape* of AE, but also inspired the invention of the technique itself. It would be interesting to attempt an implementation of AE for a different "real-world" SE system to analyze the difficulty of realizing AE *without* the concept of (abstract) updates.

Applications of Abstract Execution

Enforcing Behavioral Refactoring Preconditions We aim to increase the practical impact of our discovered behavioral preconditions for refactorings (Chapter 6) by integrating the findings into existing refactoring workflows. For example, we could add static checks for easier preconditions, like forbidden `return` statements or references to program variables, to existing refactoring routines of an IDE like Eclipse[1] of IntelliJ IDEA[2]. Additionally, the automatic addition of runtime assertions or creation of test suites asserting compliance with the preconditions is an interesting option. As a simple measure, warnings could be added to refactoring dialogs, thus increasing sensitivity of programmers regarding potential problems arising from noncompliance with behavioral preconditions.

Compilation and Optimization Abstract Execution was first mentioned in [SH18], where we proposed a rule-based compiler from Java to LLVM IR. Its distinguishing feature was that the correctness of compilation rules could be expressed as *justifying formulas* treating schematic placeholders by a simple form of AE. As AE is automatic, justifying formulas can be proven automatically. Connecting to this work, we plan to implement the SE calculus for LLVM IR introduced in [SH18], and to construct the proposed rule-based compiler with integrated, automatic verification of the (statement-based) compilation rules.

While the term "refactoring" describes behavior-preserving transformations improving a

[1] `https://www.eclipse.org/`
[2] `https://www.jetbrains.com/de-de/idea/`

"soft" property, i.e., understandability and maintainability, there are behavior-preserving transformations aiming at optimizing measurable non-functional properties such as performance. Alternatively, code can be transformed without already improving these properties, but establishing a better basis for applying actual optimizations, such as transformations to parallel pipelines [HJW14]. A research collaboration following the latter idea, which aims to use AE to prove common transformations preprocessing code to establish parallelization opportunities, has already been started. We expect to discover new challenges along that case study, in particular related to "loop-heavy" abstract program models.

Lazy SE and CbC In several case studies (e.g., [Bau+12; BB13; Gra15; Gou+19], see also [Ahr+16]) it has been discovered that not verification, but *specification* is the new bottleneck in program proving. For instance, to symbolically execute a loop, usually a sufficiently strong functional loop invariant is needed. Using external software libraries, which is generally recommendable, is a nightmare for formal verification, as libraries hardly ever come with a formal specification. A possible answer to this problem could consist in using *abstraction* to circumvent the need for specification. The principle of *Lazy* Symbolic Execution is to replace library calls, loops etc. by an APE with automatically generated frame and footprint. If nothing is known about the abstracted code fragment, frame and footprint can be overapproximate with "**`\everything`**". Frequently, however, one can find a better estimate, for instance based on the parameters of a library method call or a lightweight static analysis of a loop body. Thus, SE "lazily" runs over complicated parts of code, postponing reasoning about those to later.

Consider the small program in Fig. 8.1a computing the factorial of an integer `num`, using a print statement to output intermediate results of the computation. The method "`println`" of the static field `out` of class `System` is a library method, which has to be commented out before loading into KeY. Otherwise, SE will not finish, or the problem cannot even be loaded.[3] Lazy SE replaces the library call by an AS. By static analysis, we find out that the call to `println` cannot change any location: It is not contained in an assignment, and the passed variable is of scalar type.[4] We obtain the abstracted code shown in Fig. 8.1b. With a suitable loop invariant, this allows us to prove that after program execution, `result` will contain the factorial of `num`. If such a precise result cannot be automatically determined, we can ask the user for help: Defining the frame

[3] For this particular method, there exists a pre-defined "stub" in KeY, such that we can load this code; however, execution will stop before the call to `println`. Since for most methods in the Java standard library, there are no such stubs, most library calls will result in an error message.

[4] In fact, there is more to check, which we omit for simplicity, e.g., that the current object's class does not have *static* state which could be manipulated by the call.

```
int result = 1;


System.out.println(result);

for (int i = 2; i <= num; i++) {
  result *= i;


  System.out.println(result);
}
```

(a) Concrete Program

```
int result = 1;
//@ assignable \nothing;
//@ accessible result;
\abstract_statement Println;

for (int i = 2; i <= num; i++) {
  result *= i;
  //@ assignable \nothing;
  //@ accessible result;
  \abstract_statement Println;
}
```

(b) Abstracted Program

Figure 8.1: Example for Lazy Symbolic Execution

and footprint of a library method is still easier than specifying its *functionality*.

Usually, the effects of library calls are more important than in the above example, and contribute to the functionality of the calling code. Lazy SE is still useful: We propose a workflow consisting in dividing program proving into two "passes". First, the program is executed "lazily", leaving holes (i.e., abstract updates) in the resulting symbolic state. Secondly, these holes are filled by user-supplied specifications or automatic inference. This helps to mitigate the *path explosion problem* (cf. [CS13; Bal+18; Yan+19]) of SE and supports *incremental development of specifications* (cf. [Gou+19]). Path explosion is avoided since APEs work as *summaries*, sidestepping early branching. Incremental specification is supported since we only have to execute the main program *once*. We then can come up with loop invariants and library specifications after the fact and see whether they are good enough; otherwise, we refine them without having to start over with SE.

The concept of lazy SE can also be used in a robust approach to "top-down" programming which is *correct by construction* [KW12], working as follows: Start with a method specified with the desired contract which contains an AS with the same contract for its return behavior and a suitable abstract frame and footprint. Already now, we can prove the abstract program correct. Next, we repeatedly replace APEs in the method by legal instantiations, until no more abstract element remains. Since all instantiations are legal, all intermediate stages are correct, and could be monolithically verified. The important point, however, is that monolithic verification is not required: One just has to assert that all instantiations are legal. To that end, we need an *instantiation checker* for abstract programs assessing whether a (potentially abstract) program is an instance of an abstract program. This approach can be augmented by synthesis techniques finding existential witnesses for APEs (e.g., [Mec+18]). If at one point in the refinement loop, the problem is simple enough that the remaining holes can be synthesized, the user can stop and leave the details to automatic synthesis.

As a next step, we plan to implement an instantiation checking mechanism (cf. Sect. "Instantiation Checking"). Based on this, lazy SE can be practically evaluated. Concerning lazy SE-based CbC, we already started a research collaboration which we intend to pursue.

Abstract Cost Analysis Cost analysis "statically approximates the cost of programs in terms of their input data size" [Alb+12]. One can also analyze costs of abstract programs to perform *Abstract Cost Analysis*. Listing 8.2 shows an abstract program model with a loop containing two ASs (we omitted some details to simplify the presentation). Due to the specifications in Lines 10 and 20, the contribution of AS `Q` is constant after its first execution. Consequently, `Q` can be pulled out of the loop without changing the external (functional) behavior of the program. The abstract program model has *cost annotations* highlighted in gray. Each AS is specified to contribute an *abstract* constant amount to the program's total cost. The loop is specified with a *cost invariant* in Line 11. Together with the other invariants, we can prove that the cost of the whole program amounts to $\mathtt{threshold} \cdot (\mathit{costP}(\mathit{oldFootprintP}) + \mathit{costQ}(\mathit{oldFootprintQ}))$. Moreover, we can prove that after extraction of AS `Q`, the cost is reduced to $\mathit{costQ}(\mathit{oldFootprintQ}) + \mathtt{threshold} \cdot \mathit{costP}(\mathit{oldFootprintP})$. We can already prove the correctness of the model with cost annotations. The task of abstract cost analysis is to infer them, ideally together with sufficiently strong loop (in)variants to deduce the total cost. The model in Listing 8.2 is an intermediate result of an ongoing research collaboration with the aim to provide a fully automatic tool chain for abstract cost analysis, including verification of the correctness of cost annotations. This can also contribute to *modularity* of cost analysis: Using dynamic frames, we can "detach" a method from its context and perform (abstract) cost analysis in a modular fashion.

REFINITY Abstract strongest loop invariants are useful in abstract program equivalence proofs for the refactoring models we studied in Chapter 6. Notwithstanding, we think that the integration of "native" techniques from relational program verification should be evaluated for REFINITY, for two reasons: First, to better support verification of concrete or semi-concrete programs; and secondly, to increase scalability for larger abstract program models. To that end, the approach of LLRêve to divide programs into linear segments and use coupling invariants seems to be most promising, although some theoretic challenges

Listing 8.2: Abstract Program Model with Cost Specifications

```
/*@ ae_constraint
  @   \disjoint(frameQ, frameP) &&
  @   \disjoint(frameP, footprintQ) && // ...
  @ ; */

i = 0;

/*@ loop_invariant
  @   i >= 0 && i <= threshold && inv(\value(frameP)) &&
  @   (i > 0 ==> \value(frameQ) == resultQ(oldFootprintQ)) &&
  @   (\cost == i * (costP(oldFootprintP) + costQ(oldFootprintQ)));
  @ assignable frameP, frameQ;
  @ decreases threshold - i; */
while (i < threshold) {
    //@ ghost int oldCost = \cost;

    /*@ assignable frameQ, \hasTo(\cost);
      @ accessible footprintQ;
      @ normal_behavior ensures
      @   \value(frameQ) == resultQ(oldFootprintQ) &&
      @   \cost == oldCost + costQ(oldFootprintQ); */
    \abstract_statement Q;

    //@ set oldCost = \cost;

    /*@ assignable frameP, \hasTo(\cost);
      @ accessible footprintP;
      @ normal_behavior ensures
      @   inv(\value(frameP)) &&
      @   \cost == oldCost + costP(oldFootprintP); */
    \abstract_statement P;

    i++;
}
```

remain (see Chapter 7). Apart from that, we will add syntactic validation of abstract specifications to REFINITY and further improve its robustness and usability.

MTL / STL

We proposed STL as an independent logic mainly to reason about problems expressed in MTL. We think that symbolic traces are a natural syntactic representation of the latter, due to the small semantic gap between symbolic traces on the one side and a logic based on "Trace Description Languages" describing semantic traces on the other. To practically evaluate the feasibility of proving problems with the STL calculus, we envisage to implement it and apply it to formalizations of some of the program verification problems discussed in Sect. 5.3. One idea to compensate the calculus' incompleteness to some degree is to implement it with backtracking. Thus, it would be less problematic to remove the context when, e.g., applying a rule for sequential composition: If we chose the "wrong" pair of formulas, we can revise our choice after the spawned proof branch could not be closed. Another way to address incompleteness is to examine fragments of the symbolic trace language. For example, specifications without the choice operator are certainly less problematic. In addition, it could be worthwhile to investigate *different* "backends" for MTL. An interesting option is the first-order trace logic proposed in [Bar+19]; Projecting to trace logic would allow proving problems with existing first-order solvers.

Concerning MTL, we suggest to characterize the relation between the trace modality and asymmetric product programs [BCK13], which are conceptually near to each other. On the practical side, continuing our endeavors to formalize program verification problems and integrating existing solution techniques in a future implementation of a reasoning system for MTL is an obvious follow-up. We hope to uncover synergy potential between so far separated areas, and to unleash some of this potential in our system.

Symbolic Execution

Our work on Symbolic Execution comprises foundational aspects elaborated in Chapter 3 as well as practical components of the SE framework of the KeY system, in particular *completion scopes*. We consider adding a "progress" property for SE transition relations, which can be plugged on top of the existing properties of exhaustiveness and precision to facilitate a formal comparison to the properties suggested by other SE formalizations [Kne91; LRA17; BB19].

Together with Nathan Wasser, we will continue our work on completion scopes (**exec** statements). After implementing them for the KeY system, we aim at constructing a completion scope-based loop invariant rule for Java based on a notion of *loop contracts* also specifying the behavior of the loop in presence of abrupt completion. This nicely integrates with the concept of *strongest* abstract strongest loop invariants used in Chapter 6. We think that a loop is better specified when not ignoring its abrupt completion behavior. Finally, completion scopes can be used to master other problems in deductive verification of abruptly completing programs. Specifically, they allow constructing non-trivial product programs of abruptly completing factors, which should be explored.

9 Conclusion

In this thesis, we contributed three reasoning frameworks and one major application:

- A universal, theoretic framework for Symbolic Execution (SE) based on the semantics of *symbolic states* and m-to-n transitions,
- *Abstract Execution (AE)*, an automatic and implemented reasoning system for behavioral properties of abstract programs,
- Modal Trace Logic (MTL), a trace-based, extensible logic for formalizing a wide range of program verification problems, and *Symbolic Trace Logic (STL)*, a reasoning system for symbolic traces, and
- a case study in which we formalized standard statement-level Java refactoring techniques as abstract programs, discovered preconditions for safe refactoring applications which were, up to now, not described in literature, and proved the soundness of the transformations under those preconditions.

The central notion of SE is, in our opinion, the Symbolic Execution State (SES), a triple of a path condition, symbolic store, and program counter. It represents a set of concrete states: Given an interpretation of abstract symbols satisfying the path condition, we transform a concrete initial state according to the symbolic store and program counter. The result is called the *concretization* of the symbolic state for the interpretation and initial state. SE transitions take m SESs of a symbolic input *configuration* (a set of SE states) to n result states in the output configuration. We defined correctness properties on this idea of symbolic transitions. A transition is *precise* if any concretization of the output configuration is also a concretization of the input configuration, and is *exhaustive* if any concretization of the input configuration is also a concretization of the output configuration.

Both notions are desirable. Depending on the use case, however, generally only *one* is *indispensable*. *Heavyweight* SE is exhaustive, since anything inferred for the output states has to transfer to the original problem. *Lightweight* SE, on the other hand, is precise, to avoid false positives, e.g., in automatic bug finding or test case generation. Of course, transitions should be as precise as possible in the heavyweight case (for completeness of the reasoning system), and as exhaustive as possible in the lightweight case (to not miss any bug or test case). We formally proved the connection between exhaustiveness and program proving on the one hand, and precision and bug finding on the other hand.

Our SE framework natively supports *state merging*, a popular strategy to mitigate the *path explosion problem* of SE by merging branches in a Symbolic Execution Tree, e.g., after a case distinction induced by an **if** statement. We discuss three particular state merging techniques: If-Then-Else merging, which is exhaustive and precise, predicate abstraction-based merging, which is (generally) only exhaustive, and branch selection, which is only precise. It has been shown in a major case study [Gou+19] that our approach to state merging reduces proof sizes in practical applications, which we briefly discussed.

We do not enforce that transitions "progress", i.e., that they proceed toward a goal by reducing the program counter to changes to stores and path conditions. Neither do we couple symbolic to concrete execution, as practiced by simulation-based frameworks [LRA17; BB19]. We regard this not as a deficiency, but as expression of the *flexibility* of our framework. Progress and simulation properties could be defined "on top" of our framework, yielding an even more versatile toolbox for describing and analyzing SE systems.

"*Abstract Execution*" denotes the idea to process *Abstract* programs by symbolic *Execution*. Our framework for AE is based on a specification language for abstract programs, a deductive framework giving meaning to the latter, and, most prominently, a set of SE rules transforming Abstract Statements (ASs) and Abstract Expressions (AExps) (summarized as Abstract Program Elements (APEs)) to *abstract state changes* and *case distinctions* treating abrupt completion. With these constituents, which we implemented for the program verification platform KeY, we can *automatically* prove *behavioral* properties about abstract programs. A behavioral property regards *externally visible effects* of a program, i.e., changes to the outside state and abrupt completion, e.g., by a **return** or a thrown exception.

AE comes with two features which other *automatic* approaches to schematic program proving lack: (1) An expressive specification language for constraining locations an APE may read (its *footprint*) and write (its *frame*), when and how it completes abruptly, and which properties on the resulting state it assures after (normal or abrupt) completion, and (2) direct support of Abstract Expressions *and* Abstract Statements. With the exception of [KTL09], only one is supported. We use the theory of *dynamic frames* for frame and footprint definitions, allowing for a variable degree of abstraction: Unconstrained dynamic frame specification variables represent arbitrary location sets; by imposing additional constraints, however, we can specialize them to the point of individual locations.

The combination of expressivity and automation makes AE attractive to a plethora of applications, ranging from transformation rules over Correctness-by-Construction to abstract cost analysis. At the time of writing this thesis, three research collaborations aiming to apply AE in one of these fields have already been started; others have been contemplated. To increase the reach of AE even more, we consider implementing it in a different host framework, or to explore its potential as a general reasoning technique.

We used an early version of AE in [SH18] to modularly formalize the correctness of a rule-based compiler by projection to dynamic logic formulas with abstract programs. By now, the signature application of AE are *source-to-source* transformations. We created abstract program models of statement-level *Java refactoring rules* consisting of one abstract program representing the source, and one representing the target state of the transformed program. The first attempt to prove the model correct, i.e., to prove the two abstract programs behaviorally equivalent, normally failed: Refactoring rules have non-trivial preconditions which are usually undocumented in literature. We elicited suitable preconditions in a stepwise refinement process until we could prove the model correct. The formal preconditions we documented are precise and concise, and *guarantee behavioral equivalence* of the result with the source, i.e., the safety of the refactoring.

We developed theoretical and practical tools to support modeling and proving general relational properties of abstract programs. REFINITY is a graphical frontend for KeY for relational verification, which has been created with abstract programs in mind. It allows to conveniently specify two abstract program fragments and a relational postcondition over locations of interest. Proof obligations are created automatically and sent to KeY. We found that REFINITY significantly eased prototyping and refinement of abstract program models. The tool has been successfully used by participants of a tutorial session at iFM'19[1] to prove the correctness of a refactoring technique. To prove equivalences about abstract programs with loops, we propose *abstract strongest loop invariants*. A loop invariant is a formula which is maintained at each entry point of the loop, and can be used to abstract from the concrete behavior of the loop. A *strongest* loop invariant allows to deduce the exact effect of the loop at the program point after the loop, when combined with the precondition and the negated loop guard. The abstract version works on formulas created from abstract predicates; those are ensured by APEs inside the loop. While it is very difficult to derive strongest loop invariants for concrete programs, it is surprisingly easy to come up with *abstract* strongest ones for abstract programs, which follow a common pattern. Abrupt completion of abstract loops can also be dealt with. For this, we need an even stronger notion of invariant, which is also maintained when a loop is exited abruptly. In particular for abstract programs, loop invariants are not merely a necessary technical

[1] `https://ifm2019.hvl.no/refa/`

instrument, but also a *thinking tool* documenting the behavior of abstract programs. For example, they allow to state a description like the loop "does two different things" [Fow18] more precisely: The loop contains two ASs satisfying disjoint invariants.

Modal Trace Logic is a very flexible semantic framework with only one fixed syntactic component, the *trace modality* $[\mathscr{C}_{impl} \Vdash_{\alpha} \mathscr{C}_{spec}]$. It expresses that the specification $\mathscr{C}_{spec}$ *approximates* the implementation $\mathscr{C}_{impl}$ after the *abstraction* step defined by α. We represented several program verification problems in MTL. This requires an *instantiation step*: To represent a particular problem in MTL, decide what formal languages you need to represent the problem (e.g., Java programs and LTL formulas) and give them a *trace semantics*. We call such formal languages with trace semantics *Trace Description Languages (TDLs)*. Then, choose a suitable (combination of) trace abstractions, for instance, big-step abstraction and restriction to a set of interesting locations. For our formalizations, we defined some re-occurring TDLs and abstractions. Frequently, it is sufficient to pick and combine abstractions from a predefined set of commonly used ones. Notwithstanding, our framework is flexible enough to account for more specialized definitions, such as the "patch abstraction" we stipulated to represent program repair.

One particular TDL we use is the language of *abstract programs*. For instance, in program synthesis or rule-based compilation, unknown parts of a program are represented by schematic statements. Abstract programs are, apart from *approximation* and *abstraction*, the third pillar on which MTL and the trace modality are built. Together, those enabled us to succinctly represent diverse verification tasks.

To facilitate reasoning about problems expressed in MTL, we translate them to *regular symbolic traces* and use Symbolic Trace Logic, a logic for reasoning about inclusion of trace sets represented by symbolic traces. Translations from MTL to STL are conditioned, allowing to conclude from a proof of an STL judgment $\mathfrak{a}(\varpi_{spec}) \vdash \mathfrak{a}(\varpi_{impl})$ the universal validity of an MTL assertion $[\mathscr{C}_{impl} \Vdash_{\alpha} \mathscr{C}_{spec}]$. STL's symbolic trace language is based on the fundamental notion of symbolic states and their concretization-based semantics. The core component of its *sequent calculus* is a *subsumption checker* for symbolic states. We formally defined two generally useful notions of *weak* and *strong subsumption* of symbolic states. We *proved the soundness* of the STL calculus, which is, on the other hand, incomplete. This motivates future work on translating MTL to complete systems, e.g., trace logics embedded in first-order logic [Bar+19]. The calculus is still attractive: It consists of a relatively small set of easily understandable rules, which can be modularly analyzed, e.g., for soundness. The soundness argument for the automata-based approach we proposed in [SH19b], which involved rather complicated constructions, was much more intricate, and not conducted on the same formal level as our soundness proof for STL.

References

[ARS05] Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. “Automatic Validation of Transformation Rules for Java Verification Against a Rewriting Semantics”. In: *Proc. 12th International Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Geoff Sutcliffe and Andrei Voronkov. Vol. 3835. Lecture Notes in Computer Science. Springer, 2005, pp. 412–426.

[Ahr+16] Wolfgang Ahrendt et al., eds. *Deductive Software Verification – The KeY Book*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016.

[Alb+12] Elvira Albert et al. “Cost Analysis of Object-Oriented Bytecode Programs”. In: *Theor. Comput. Sci.* 413.1 (2012), pp. 142–159.

[Alk+10] Eyad Alkassar et al. “Automated Verification of a Small Hypervisor”. In: *Proc. Third International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. Ed. by Gary T. Leavens, Peter W. O’Hearn, and Sriram K. Rajamani. Vol. 6217. Lecture Notes in Computer Science. Springer, 2010, pp. 40–54.

[AML17] Everton L. G. Alves, Tiago Massoni, and Patrıcia Duarte de Lima Machado. “Test Coverage of Impacted Code Elements for Detecting Refactoring Faults: An Exploratory Study”. In: *Journal of Systems and Software* 123 (2017), pp. 223–238.

[APV06] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. “Symbolic Execution with Abstract Subsumption Checking”. In: *13th Intl. Conf. on Model Checking Software*. Springer, 2006.

[App12] Andrew W. Appel. “Verified Software Toolchain”. In: *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Ed. by Alwyn Goodloe and Suzette Person. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012, p. 2.

[Avg+14] Thanassis Avgerinos et al. “Automatic Exploit Generation”. In: *Commun. ACM* 57.2 (Feb. 2014), pp. 74–84. issn: 0001-0782.

[Bal+18] Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *ACM Comput. Surv.* 51.3 (2018), 50:1–50:39.

[BNN16] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. “Relational Logic with Framing and Hypotheses”. In: *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*. Ed. by Akash Lal et al. Vol. 65. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 11:1–11:16.

[BL05] Mike Barnett and K Rustan M Leino. “Weakest-Precondition of Unstructured Programs”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 31. ACM. 2005, pp. 82–87.

[Bar+05] Mike Barnett et al. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 364–387.

[BT07] Clark W. Barrett and Cesare Tinelli. “CVC3”. In: *Proc. 19th International Conf. on Computer Aided Verification*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 298–302.

[BCK13] Gilles Barthe, Juan Manuel Crespo, and César Kunz. “Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification”. In: *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*. Ed. by Sergei N. Artëmov and Anil Nerode. Vol. 7734. Lecture Notes in Computer Science. Springer, 2013, pp. 29–43.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. “Relational Verification Using Product Programs”. In: *17th Intl. Symp. on Formal Methods (FM)*. Ed. by Michael J. Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214.

[BDR04] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. “Secure Information Flow by Self-Composition”. In: *17th IEEE Computer Security Foundations Workshop, CSFW-17*. IEEE Computer Society, 2004, pp. 100–114.

[Bar+19] Gilles Barthe et al. “Verifying Relational Properties using Trace Logic”. In: *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*. Ed. by Clark W. Barrett and Jin Yang. IEEE, 2019, pp. 170–178.

[Bau+12] Christoph Baumann et al. “Lessons Learned From Microkernel Verification – Specification is the New Bottleneck”. In: *Proc. 7th Conf. on Systems Software Verification (SSV)*. Ed. by Franck Cassez et al. Vol. 102. EPTCS. 2012, pp. 18–32.

[BB13] Bernhard Beckert and Daniel Bruns. “Dynamic Logic with Trace Semantics”. In: *Proc. CADE-24*. 2013, pp. 315–329.

[BU18] Bernhard Beckert and Mattias Ulbrich. “Trends in Relational Program Verification”. In: *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*. 2018, pp. 41–58.

[Bec+17] Bernhard Beckert et al. “Proving JDK’s Dual Pivot Quicksort Correct”. In: *Revised Selected Papers of the 9th Intern. Conf. on Verified Software. Theories, Tools, and Experiments VSTTE*. Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Lecture Notes in Computer Science. Springer, 2017, pp. 35–48.

[Ben04] Nick Benton. “Simple Relational Correctness Proofs for Static Analyses and Program Transformations”. In: *Proc. 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 14–25.

[Ber11] Lennart Beringer. “Relational Decomposition”. In: *2nd International Conference on Interactive Theorem Proving (ITP)*. Ed. by Marko C. J. D. van Eekelen et al. Vol. 6898. Lecture Notes in Computer Science. Springer, 2011, pp. 39–54.

[Bie+03] Armin Biere et al. “Bounded Model Checking”. In: *Advances in Computers* 58 (2003), pp. 117–148.

[Bob+08] Francois Bobot et al. *The Alt-Ergo Automated Theorem Prover*. 2008. url: `http://alt-ergo.lri.fr`.

[Bob+11] François Bobot et al. “Why3: Shepherd Your Herd of Provers”. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. 2011, pp. 53–64.

[BB19] Frank S. de Boer and Marcello M. Bonsangue. “On the Nature of Symbolic Execution”. In: *Proc. 3rd World Congress on Formal Methods (FM)*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 64–80.

[BMR95] Alexander Borgida, John Mylopoulos, and Raymond Reiter. “On the Frame Problem in Procedure Specifications”. In: *IEEE Trans. Software Eng.* 21.10 (1995), pp. 785–798.

[BDP15] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. “Symbolic Execution of Programs with Heap Inputs”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ACM, 2015, pp. 602–613.

[BHH19] Richard Bubel, Reiner Hähnle, and Asmae Heydari Tabar. “A Program Logic For Dependence Analysis”. In: *Proc. 15th Intern. Conf on Integrated Formal Methods (IFM)*. Ed. by Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa. Lecture Notes in Computer Science. Springer, 2019.

[BHP14] Richard Bubel, Reiner Hähnle, and Maria Pelevina. “Fully Abstract Operation Contracts”. In: *Proc. 6th Intern. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 8803. Lecture Notes in Computer Science. Springer, 2014, pp. 120–134.

[BRR08] Richard Bubel, Andreas Roth, and Philipp Rümmer. “Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic”. In: *Electr. Notes Theor. Comput. Sci.* 199 (2008), pp. 107–128.

[Bur74] Rodney Martineau Burstall. “Program Proving as Hand Simulation with a Little Induction”. In: *Information Processing*. Elsevier, 1974, pp. 308–312.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”. In: *8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[CS13] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Communications of the ACM* 56.2 (2013), pp. 82–90. issn: 0001-0782.

[Cad+06] Cristian Cadar et al. “EXE: Automatically Generating Inputs of Death”. In: *13th ACM Conference on Computer and Communications Security, (CCS)*. Ed. by Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati. ACM, 2006, pp. 322–335.

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “The S2E Platform: Design, Implementation, and Applications”. In: *ACM Trans. Comput. Syst.* 30.1 (2012), 2:1–2:49.

[CJM14] Duc-Hiep Chu, Joxan Jaffar, and Vijayaraghavan Murali. "Lazy Symbolic Execution for Enhanced Learning". In: *Proc. of the 5th Intl. Conf. on Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Springer, 2014, pp. 323–339.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Proc. TACAS 2004*. 2004, pp. 168–176.

[Cok14] David R. Cok. "OpenJML: Software Verification for Java 7 Using JML, OpenJDK, and Eclipse". In: *Proc. 1st Workshop on Formal Integrated Development Environments*. 2014, pp. 79–92.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *4th Symp. of POPL*. ACM Press, Jan. 1977, pp. 238–252.

[Cra57a] William Craig. "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem". In: *J. Symb. Log.* 22.3 (1957), pp. 250–268.

[Cra57b] William Craig. "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory". In: *J. Symb. Log.* 22.3 (1957), pp. 269–285.

[Cuo+12] Pascal Cuoq et al. "Frama-C". In: *Proc. Intern. Conf. on Software Engineering and Formal Methods*. Springer. 2012, pp. 233–247.

[Dah+09] Markus Dahlweid et al. "VCC: Contract-Based Modular Verification of Concurrent C". In: *31st Intern. Conf. on Software Engineering (ICSE)*. IEEE, 2009, pp. 429–430.

[Dan+07] Brett Daniel et al. "Automated Testing of Refactoring Engines". In: *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2007, pp. 185–194.

[DE82] R.B. Dannenberg and G.W. Ernst. "Formal Program Verification Using Symbolic Execution". In: *IEEE Transactions on Software Engineering* SE-8.1 (1982), pp. 43–52. issn: 0098-5589.

[DHS05] Ádám Darvas, Reiner Hähnle, and David Sands. "A Theorem Proving Approach to Analysis of Secure Information Flow". In: *Proc. 2nd Intern. Conf. on SPC*. 2005, pp. 193–209.

[DV13] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *Proc. 23rd IJCAI*. 2013, pp. 854–860.

[DLR06] Xianghua Deng, Jooyong Lee, and Robby. "Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems". In: *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE '06*. 2006, pp. 157–166.

[Din+17] Crystal Chang Din et al. "Locally Abstract, Globally Concrete Semantics of Concurrent Programming Languages". In: *Proc. 26th TABLEAUX*. 2017, pp. 22–43.

[Dow+93] Gilles Dowek et al. *The Coq Proof Assistant User's Guide*. Rapport Techniques 154. INRIA-Rocquencourt, France, 1993.

[Dre19] Benedikt Dreher. "Transparent Treatment of Loops in JavaDL". Bachelor's Thesis. Technische Universität Darmstadt, Germany, 2019.

[EBS16] Anna Maria Eilertsen, Anya Helene Bagge, and Volker Stolz. "Safer Refactorings". In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9952. Lecture Notes in Computer Science. 2016, pp. 517–531.

[ED07] Dawson R. Engler and Daniel Dunbar. "Under-Constrained Execution: Making Automatic Code Destruction Easy and Scalable". In: *Proc. of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Ed. by David S. Rosenblum and Sebastian G. Elbaum. ACM, 2007, pp. 1–4.

[Fil11] Jean-Christophe Filliâtre. "Deductive Software Verification". In: *International Journal on Software Tools for Technology Transfer (STTT)* 13.5 (2011), pp. 397–403.

[Fit07] Melvin Fitting. "Modal Proof Theory". In: *Handbook of Modal Logic*. Ed. by Patrick Blackburn, J. F. A. K. van Benthem, and Frank Wolter. Vol. 3. Studies in logic and practical reasoning. North-Holland, 2007, pp. 85–138.

[FQ02] Cormac Flanagan and Shaz Qadeer. "Predicate Abstraction for Software Verification". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL '02. New York, NY, USA: ACM, 2002, pp. 191–202.

[FS01] Cormac Flanagan and James B. Saxe. "Avoiding Exponential Explosion: Generating Compact Verification Conditions". In: *SIGPLAN Not.* 36.3 (Jan. 2001), pp. 193–205. issn: 0362-1340.

[Fla+02] Cormac Flanagan et al. “Extended Static Checking for Java”. In: *SIGPLAN Not.* 37.5 (May 2002), pp. 234–245. issn: 0362-1340.

[Flo67] Robert W Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science*. Proc. Symp. in Applied Mathematics 19 (1967), pp. 19–32.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, June 1999.

[Fow18] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series. 2nd edition. Addison-Wesley Professional, Nov. 2018.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automated Theorem Proving*. Harper and Row, New York, 1986.

[GM06] Alejandra Garrido and Jose Meseguer. “Formal Specification and Verification of Java Refactorings”. In: *Proc. 6th IEEE Intern. Workshop on Source Code Analysis and Manipulation*. SCAM ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 165–174.

[Gen35] Gerhard Gentzen. “Untersuchungen über das Logische Schließen”. In: *Mathematische Zeitschrift* 39 (1935), pp. 176–210, 405–431.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing”. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 213–223.

[Göd31] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für Mathematik und Physik* 38.1 (1931), pp. 173–198.

[GS13] Benny Godlin and Ofer Strichman. “Regression Verification: Proving the Equivalence of Similar Programs”. In: *Softw. Test., Verif. Reliab.* 23.3 (2013), pp. 241–258.

[Gos+05] James Gosling et al. *The Java (TM) Language Specification*. 3rd. Addison-Wesley Professional, 2005. isbn: 0321246780.

[GBR14] Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. “Proof Pearl: The KeY to Correct and Stable Sorting”. In: *J. Autom. Reasoning* 53.2 (2014), pp. 129–139.

[Gou+19] Stijn de Gouw et al. "Verifying OpenJDK's Sort Method for Generic Collections". In: *J. Autom. Reasoning* 62.1 (2019), pp. 93–126.

[Gou+15] Stijn de Gouw et al. "OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case". In: *Proc. of the 27th Intl. Conf. on Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Pasareanu. Springer, 2015.

[GS97] Susanne Graf and Hassen Saidi. "Construction of Abstract State Graphs with PVS". In: *Proc. of the 9th Intl. Conf. on Computer Aided Verification*. Ed. by Orna Grumberg. Springer, 1997, pp. 72–83.

[Gra15] Daniel Grahl. "Deductive Verification of Concurrent Programs and its Application to Secure Information Flow for Java". PhD thesis. Karlsruhe Institute of Technology, 2015.

[Häh+86] R. Hähnle et al. "An Interactive Verification System based on Dynamic Logic". In: *8th CADE*. Ed. by Jörg H. Siekmann. Lecture Notes in Computer Science 230. Springer, 1986, pp. 306–315.

[HH19] Reiner Hähnle and Marieke Huisman. "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools". In: *Computing and Software Science - State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard J. Woeginger. Vol. 10000. Lecture Notes in Computer Science. Springer, 2019, pp. 345–373.

[HT08] Jonathan de Halleux and Nikolai Tillmann. "Parameterized Unit Testing with Pex". In: *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*. Ed. by Bernhard Beckert and Reiner Hähnle. Vol. 4966. Lecture Notes in Computer Science. Springer, 2008, pp. 171–181.

[HSS09] Trevor Hansen, Peter Schachte, and Harald Søndergaard. "State Joining and Splitting for the Symbolic Execution of Binaries". In: *Proc. of the 9th Intl. Workshop on Runtime Verification*. Ed. by Saddek Bensalem and Doron A. Peled. Springer, 2009, pp. 76–92.

[HTK00] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. Cambridge, MA, USA: MIT Press, 2000. isbn: 0262082896.

[Haw+13] Chris Hawblitzel et al. "Towards Modularly Comparing Programs Using Automated Theorem Provers". In: *24th International Conference on Automated Deduction (CADE)*. Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 282–299.

[HM03] Görel Hedin and Eva Magnusson. “JastAdd–An Aspect-Oriented Compiler Construction System”. In: *Sci. Comput. Program.* 47.1 (2003), pp. 37–58.

[Hei92] Maritta Heisel. “Formalizing and Implementing Gries’ Program Development Method in Dynamic Logic”. In: *Sci. Comput. Program.* 18.1 (1992), pp. 107–137.

[HHB14] Martin Hentschel, Reiner Hähnle, and Richard Bubel. “Visualizing Unbounded Symbolic Execution”. In: *Tests and Proofs*. Ed. by Martina Seidl and Nikolai Tillmann. Lecture Notes in Computer Science 8570. Springer International Publishing, 2014, pp. 82–98.

[Hoa69] Charles Antony Richard Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Hol97] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295.

[Hov12] Dag Hovland. “The Inclusion Problem for Regular Expressions”. In: *J. Comput. Syst. Sci.* 78.6 (2012), pp. 1795–1813.

[HJW14] Zia Ul Huda, Ali Jannesari, and Felix Wolf. “Using Template Matching to Infer Parallel Design Patterns”. In: *TACO* 11.4 (2014), 64:1–64:21.

[HJ00] Marieke Huisman and Bart Jacobs. “Java Program Verification via a Hoare Logic with Abrupt Termination”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2000, pp. 284–303.

[Jac06] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. isbn: 978-0-262-10114-1.

[JMN13] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. “Boosting Concolic Testing via Interpolation”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, USA: ACM, 2013, pp. 48–58.

[Jaf+12] Joxan Jaffar et al. “TRACER: A Symbolic Execution Tool for Verification”. In: *Proceedings of the 24th International Conference on Computer Aided Verification*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 758–766.

[Jam+12] Konrad Jamrozik et al. “Augmented Dynamic Symbolic Execution”. In: *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. Ed. by Michael Goedicke, Tim Menzies, and Motoshi Saeki. Essen, Germany: IEEE, 2012, pp. 254–257.

[JM09] Ranjit Jhala and Rupak Majumdar. "Software Model Checking". In: *ACM Comput. Surv.* 41.4 (2009), 21:1–21:54.

[Kam19] Eduard Kamburjan. "Behavioral Program Logic". In: *28th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*. Ed. by Serenella Cerrito and Andrei Popescu. Vol. 11714. Lecture Notes in Computer Science. Springer, 2019, pp. 391–408.

[Kas06] Ioannis T. Kassios. "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions". In: *Proc. 14th Intern. Symposium on Formal Methods (FM)*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 268–283.

[Kas11] Ioannis T. Kassios. "The Dynamic Frames Theory". In: *Formal Asp. Comput.* 23.3 (2011), pp. 267–288.

[KKU18] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. "Relational Program Reasoning Using Compiler IR - Combining Static Verification and Dynamic Analysis". In: *J. Autom. Reasoning* 60.3 (2018), pp. 337–363.

[Kin76] James C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394. issn: 0001-0782.

[KN06] Gerwin Klein and Tobias Nipkow. "A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler". In: *ACM Trans. PLS* 28.4 (July 2006), pp. 619–695. issn: 0164-0925.

[Kle+10] Gerwin Klein et al. "seL4: Formal Verification of an Operating-System Kernel". In: *Commun. ACM* 53.6 (2010), pp. 107–115.

[Kne91] Ralf Kneuper. "Symbolic Execution: A Semantic Approach". In: *Sci. Comput. Program.* 16.3 (1991), pp. 207–249.

[KW12] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012. isbn: 978-3-642-27918-8.

[Kum+14] Ramana Kumar et al. "CakeML: A Verified Implementation of ML". In: *41st POPL*. 2014, pp. 179–192.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. "Proving Optimizations Correct Using Parameterized Program Equivalence". In: *Proc. PLDI 2009*. 2009, pp. 327–337.

[Kuz+12] Volodymyr Kuznetsov et al. "Efficient State Merging in Symbolic Execution". In: *Proc. of the 33rd Conf. on PLDI*. ACM, 2012, pp. 193–204.

[Lah+12] Shuvendu K. Lahiri et al. “SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs”. In: *24th International Conference on Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 712–717.

[Lan18] Florian Lanzinger. “A Divide-and-Conquer Strategy with Block and Loop Contracts for Deductive Program Verification”. Bachelor’s Thesis. Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Apr. 2018.

[Lea+13] Gary T. Leavens et al. *JML Reference Manual*. Draft revision 2344. May 2013.

[Lei10] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Proc. 16th Intern. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370.

[Lei05] K. Rustan M. Leino. “Efficient Weakest Preconditions”. In: *Information Processing Letters* 93.6 (2005), pp. 281–288. issn: 0020-0190.

[Ler09] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[Lon72] Ralph L. London. “Correctness of a Compiler for a Lisp Subset”. In: *Prof. of ACM Conf. on Proving Assertions About Programs*. ACM, 1972, pp. 121–127.

[Lop+18] Nuno P. Lopes et al. “Practical Verification of Peephole Optimizations with Alive”. In: *Commun. ACM* 61.2 (2018), pp. 84–91.

[LRA17] Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. “A Generic Framework for Symbolic Execution: A Coinductive Approach”. In: *J. Symb. Comput.* 80 (2017), pp. 125–163.

[Lyn59] Roger C Lyndon. “An Interpolation Theorem in the Predicate Calculus”. In: *Pacific Journal of Mathematics* 9.1 (1959), pp. 129–142.

[Ma+11] Kin-Keung Ma et al. “Directed Symbolic Execution”. In: *Proc. 18th Intern. Symp. on Static Analysis*. Ed. by Eran Yahav. Vol. 6887. Lecture Notes in Computer Science. Springer, 2011, pp. 95–111.

[MPU04] C. Marché, C. Paulin-Mohring, and X. Urbain. “The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs Annotated in JML”. In: *The Journal of Logic and Algebraic Programming* 58.1–2 (2004), pp. 89–106. issn: 1567-8326.

[McC63] John McCarthy. “A Basis for a Mathematical Theory of Computation”. In: *Computer Programming and Formal Systems*. Ed. by P. Braffort and D. Hirschberg. North Holland, 1963, pp. 33–69.

[MP67] John McCarthy and James Painter. “Correctness of a Compiler for Arithmetic Expressions”. In: *Mathematical Aspects of Computer Science* 1 (1967).

[Mec+18] Sergey Mechtaev et al. “Symbolic Execution with Existential Second-Order Constraints”. In: *Proc. 2018 Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*. 2018, pp. 389–399.

[Men17] Melissa Mendoza. “Merge Block Contracts for a Dynamic Logic Calculus”. Bachelor’s Thesis. Technische Universität Darmstadt, Germany, 2017.

[MR06] José Meseguer and Grigore Rosu. “The Rewriting Logic Semantics Project”. In: *Electr. Notes Theor. Comput. Sci.* 156.1 (2006), pp. 27–56.

[MW91] José Meseguer and Timothy C. Winkler. “Parallel Programmming in Maude”. In: *Research Directions in High-Level Parallel Programming Languages*. Ed. by Jean-Pierre Banâtre and Daniel Le Métayer. Vol. 574. Lecture Notes in Computer Science. Springer, 1991, pp. 253–293.

[MS72] Albert R. Meyer and Larry J. Stockmeyer. “The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space”. In: *13th Annual Symposium on Switching and Automata Theory*. Ed. by Allan B. Borodin et al. IEEE Computer Society, 1972, pp. 125–129.

[Mon18] Martin Monperrus. “Automatic Software Repair: A Bibliography”. In: *ACM Comput. Surv.* 51.1 (2018), 17:1–17:24.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. “How Are Java Software Developers Using the Eclipse IDE?” In: *IEEE Software* 23.4 (2006), pp. 76–83.

[MPB12] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. “How We Refactor, and How We Know It”. In: *IEEE Trans. Software Eng.* 38.1 (2012), pp. 5–18.

[NZ13] Kedar S. Namjoshi and Lenore D. Zuck. “Witnessing Program Transformations”. In: *20th International Symposium on Static Analysis (SAS)*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 304–323.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer New York, Inc., 1999. isbn: 3540654100.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. isbn: 3-540-43376-7.

[PL10] Dillon Pariente and Emmanuel Ledinot. “Formal Verification of Industrial C Code using Frama-C: A Case Study”. In: *Proc. of the 1st Intl. Conf. on FoVeOOS* (2010), p. 205.

[PV04] Corina S. Pasareanu and Willem Visser. “Verification of Java Programs Using Symbolic Execution and Invariant Generation”. In: *Proc. 11th Intern. SPIN Workshop on Model Checking Software*. 2004, pp. 164–181.

[RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *22nd POPL*. 1995, pp. 49–61.

[RV99] Alexandre Riazanov and Andrei Voronkov. “Vampire”. In: *Proc. 16th Intern. Conference on Automated Deduction (CADE)*. Ed. by Harald Ganzinger. Vol. 1632. Lecture Notes in Computer Science. Springer, 1999, pp. 292–296.

[RS10] Grigore Rosu and Traian-Florin Serbanuta. “An Overview of the K Semantic Framework”. In: *J. Log. Algebr. Program.* 79.6 (2010), pp. 397–434.

[SM03] Andrei Sabelfeld and Andrew C. Myers. “A Model for Delimited Information Release”. In: *Proc. 2nd Intern. Symp. on Software Security - Theories and Systems*. 2003, pp. 174–191.

[Sch+12] Max Schäfer et al. “A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs”. In: *IEEE Trans. Software Eng.* 38.6 (2012), pp. 1233–1257.

[Sch15] Dominic Scheurer. “From Trees to DAGs: A General Lattice Model for Symbolic Execution”. **Note:** The author D. Scheurer is the author of this thesis, who changed his name to Steinhöfel in 2017. Master’s Thesis. Technische Universität Darmstadt, 2015.

[SHB16] Dominic Scheurer, Reiner Hähnle, and Richard Bubel. "A General Lattice Model for Merging Symbolic Execution Branches". In: *Proc. 18th ICFEM*. Ed. by Kazuhiro Ogata, Mark Lawford, and Shaoying Liu. Vol. 10009. Lecture Notes in Computer Science. **Note:** The author D. Scheurer is the author of this thesis, who changed his name to Steinhöfel in 2017. Springer, 2016, pp. 57–73.

[SUW11] Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. "Dynamic Frames in Java Dynamic Logic". In: *Intl. Conf. on Formal Verification of Object-Oriented Software (FoVeOOS)*. Ed. by Bernhard Beckert and Claude Marché. Vol. 6528. Lecture Notes in Computer Science. Springer, 2011, pp. 138–152.

[Sen+15] Koushik Sen et al. "MultiSE: Multi-Path Symbolic Execution using Value Summaries". In: *10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 842–853.

[Sha18] Natarajan Shankar. "Combining Model Checking and Deduction". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 651–684.

[SSM15] Tarciana Dias da Silva, Augusto Sampaio, and Alexandre Mota. "Verifying Transformations of Java Programs Using Alloy". In: *18th Brazilian Symposium on Formal Methods: Foundations and Applications (SBMF)*. Ed. by Márcio Cornélio and Bill Roscoe. Vol. 9526. Lecture Notes in Computer Science. Springer, 2015, pp. 110–126.

[Smi90] Douglas R. Smith. "KIDS: A Semiautomatic Program Development System". In: *IEEE Trans. Software Eng.* 16.9 (1990), pp. 1024–1043.

[SGM13] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. "Automated Behavioral Testing of Refactoring Engines". In: *IEEE Trans. Software Eng.* 39.2 (2013), pp. 147–162.

[Soa+11] Gustavo Soares et al. "Analyzing Refactorings on Software Repositories". In: *25th Brazilian Symposium on Software Engineering (SBES)*. IEEE Computer Society, 2011, pp. 164–173.

[Soa+10] Gustavo Soares et al. "Making Program Refactoring Safer". In: *IEEE Software* 27.4 (2010), pp. 52–57.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. "From Program Verification to Program Synthesis". In: *Proc. 37th POPL*. 2010, pp. 313–326.

[SH19a] Dominic Steinhöfel and Reiner Hähnle. “Abstract Execution”. In: *Proc. Third World Congress on Formal Methods - The Next 30 Years, (FM)*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 319–336.

[SH18] Dominic Steinhöfel and Reiner Hähnle. “Modular, Correct Compilation with Automatic Soundness Proofs”. In: *Proc. 8th ISOLA*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. 2018.

[SH19b] Dominic Steinhöfel and Reiner Hähnle. “The Trace Modality”. In: *Dynamic Logic. New Trends and Applications - Second International Workshop (DaLi)*. Ed. by Luıs Soares Barbosa and Alexandru Baltag. Vol. 12005. Lecture Notes in Computer Science. Springer, 2019, pp. 124–140.

[SW17] Dominic Steinhöfel and Nathan Wasser. “A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows”. In: *Proc. 13th Intern. Conf on Integrated Formal Methods (IFM)*. Ed. by Nadia Polikarpova and Steve Schneider. Vol. 10510. Lecture Notes in Computer Science. Springer, 2017, pp. 279–294.

[Ste05] Kurt Stenzel. “Verification of Java Card Programs”. PhD thesis. University of Augsburg, Germany, 2005.

[Str02] Martin Strecker. “Formal Verification of a Java Compiler in Isabelle”. In: *Proc. 18th Intern. Conf. on Automated Deduction (CADE)*. Ed. by Andrei Voronkov. Vol. 2392. Lecture Notes in Computer Science. Springer, 2002, pp. 63–77.

[Tan+16] Yong Kiam Tan et al. “A New Verified Compiler Backend for CakeML”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. ACM, 2016, pp. 60–73.

[Coq19] The Coq Development Team. *The Coq Proof Assistant, version 8.10.0*. Version 8.10.0. Oct. 2019.

[TS96] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Vol. 43. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996. isbn: 978-0-521-57223-1.

[Tue10] Thomas Tuerk. “Local Reasoning about While-Loops”. In: *Third International Conference on Verified Software: Theories, Tools, Experiments (VSTTE), Theory Workshop, Proceedings*. Ed. by Rajeev Joshi et al. ETH Zürich, 2010, pp. 29–39.

[VJB12] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. “Equivalence Checking of Static Affine Programs using Widening to Handle Recurrences”. In: *ACM Trans. Program. Lang. Syst.* 34.3 (2012), 11:1–11:35.

[Vis+03] Willem Visser et al. “Model Checking Programs”. In: *Autom. Softw. Eng.* 10.2 (2003), pp. 203–232.

[VJP15] Frédéric Vogels, Bart Jacobs, and Frank Piessens. “Featherweight VeriFast”. In: *Logical Methods in Computer Science* 11.3 (2015).

[Wac12] Simon Wacker. “Blockverträge”. Studienarbeit. Karlsruhe Institute of Technology, 2012.

[Was16] Nathan Wasser. “Automatic Generation of Specifications using Verification Tools”. PhD thesis. Darmstadt: Technische Universität Darmstadt, Jan. 2016.

[WS19] Nathan Wasser and Dominic Steinhöfel. “Technical Report: Using Loop Scopes with for-Loops”. In: *CoRR* abs/1901.06839 (2019). arXiv: `1901.06839`.

[WS20] Nathan Wasser and Dominic Steinhöfel. “Treating for-Loops as First-Class Citizens in Proofs”. In: *CoRR* abs/2002.00776 (2020). arXiv: `2002.00776`.

[Xie+05] Tao Xie et al. “Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution”. In: *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 365–381.

[Yan+19] Guowei Yang et al. “Advances in Symbolic Execution”. In: *Advances in Computers*. Ed. by Atif M. Memon. Vol. 113. Advances in Computers. Elsevier, 2019, pp. 225–287.

[Yan07] Hongseok Yang. “Relational Separation Logic”. In: *Theoretical CS* 375.1-3 (2007), pp. 308–334.

A Loop Scope Invariant Rule: Statistics and Taclet

The results of our new empirical evaluation of the performance of the loop scope invariant rule are shown in Table A.1 (without one-step simplification) and Table A.2 (with one-step simplification).

Our taclet implementation for the loop scope invariant rule is shown below in Listing A.1. The depicted version is for the diamond modality. A "free invariant", which occurs with placeholder `freeInv` in the taclet, is a formula which is assumed to be an invariant, but does not have to be proven. Implementing the rule as a taclet made this explicit, and also unveiled a mistake in the previous implementation, where the free invariant had to be proven, contradicting the overall idea.

Listing A.1: Loop Scope Invariant Rule Taclet for the Diamond Modality

```
loopScopeInvDia {
  \schemaVar \formula inv;
  \schemaVar \formula freeInv;
  \schemaVar \term any variantTerm;
  \schemaVar \formula loopFormula;
  \schemaVar \program Statement #loopStmt;
  \schemaVar \program Variable #variant;

  \schemaVar \skolemTerm Heap anon_heap_LOOP;
  \schemaVar \skolemTerm Heap anon_savedHeap_LOOP;
  \schemaVar \skolemTerm Heap anon_permissions_LOOP;

  \schemaVar \program Variable #heapBefore_LOOP;
  \schemaVar \program Variable #savedHeapBefore_LOOP;
  \schemaVar \program Variable #permissionsBefore_LOOP;

  \find ( (\modality{#dia} {.. while (#nse) #body ... }\endmodality(post)) )

```

```
\varcond(\new(#x, boolean))
\varcond(\new(#variant, any))
\varcond(\new(#heapBefore_LOOP, Heap))
\varcond(\new(#savedHeapBefore_LOOP, Heap))
\varcond(\new(#permissionsBefore_LOOP, Heap))

\varcond(\storeTermIn(loopFormula,
  \modality{#dia}{ while (#nse) #body }\endmodality(post)))
\varcond(\storeStmtIn(#loopStmt,
  \modality{#dia}{ while (#nse) #body }\endmodality(post)))
\varcond(\hasInvariant(#loopStmt, #dia))
\varcond(\getInvariant(#loopStmt, #dia, inv))
\varcond(\getFreeInvariant(#loopStmt, #dia, freeInv))
\varcond(\getVariant(#loopStmt, variantTerm))

"Invariant␣Initially␣Valid":
  \replacewith(inv);

"Invariant␣Preserved␣and␣Used":
  \replacewith (
    {   #createAbstractAnonUpdate(loopFormula)
     || #createBeforeLoopUpdate(loopFormula, #heapBefore_LOOP,
          #savedHeapBefore_LOOP, #permissionsBefore_LOOP)
     || #createLocalAnonUpdate(loopFormula)
     || #createHeapAnonUpdate(loopFormula, anon_heap_LOOP,
          anon_savedHeap_LOOP, anon_permissions_LOOP)}
      {#variant:=variantTerm}
        (inv & freeInv ->
           (\modality{#dia}{
              ..
              boolean #x;
              loop-scope(#x) {
                if (#nse) {
                  #body
                  continue;
                } else {
                  break;
                }
              }
              ...
            }\endmodality(
              (#x<<loopScopeIndex>> = TRUE -> post) &
```

```
                (#x<<loopScopeIndex>> = FALSE ->
                    inv
                  & #createFrameCond(loopFormula, #heapBefore_LOOP,
                      #savedHeapBefore_LOOP, #permissionsBefore_LOOP)
                  & prec(variantTerm, #variant))
               )))
    )

    \add (#wellFormedCond(loopFormula, anon_heap_LOOP,
      anon_savedHeap_LOOP, anon_permissions_LOOP) ==>)

  \heuristics(loop_scope_inv_taclet)
};
```

Problem	Proof Nodes Old	Proof Nodes New	% Diff. # Nodes	# SE Steps Old	# SE Steps New	% Diff. # SE Steps
coincidence_count/project	14,517	27,907	*-92.24%*	211	152	27.96%
removeDups/removeDup	20,016	21,293	*-6.38%*	369	309	16.26%
list_seq/ArrayList.remove.1	12,986	13,646	*-5.08%*	260	196	24.62%
list/LinkedList_get_normal	7,063	7,395	*-4.70%*	182	170	6.59%
java_dl/jml-information-flow	49,176	50,326	*-2.34%*	475	417	12.21%
list/ArrayList_concatenate	23,045	22,684	1.57%	640	568	11.25%
vstte10_03_LinkedList/Node_search	7,730	7,562	2.17%	97	59	39.18%
SmansEtAl/ArrayList_add	6,282	6,021	4.15%	435	411	5.52%
list_recursiveSpec/...setValueAt	5,036	4,804	4.61%	184	154	16.30%
saddleback_search/Saddleback_search	32,875	30,632	6.82%	236	182	22.88%
observer/ExampleSubject_addObserver	4,757	4,413	7.23%	167	136	18.56%
list_ghost/ArrayList_enlarge	2,715	2,517	7.29%	151	128	15.23%
WeideEtAl_02_BinarySearch/BinarySearch_search	4,518	4,169	7.72%	182	147	19.23%
list/ArrayList_enlarge	3,181	2,927	7.98%	156	133	14.74%
vacid0_01_SparseArray/MemoryAllocator_alloc	1,076	990	7.99%	89	78	12.36%
list_seq/ArrayList.enlarge	3,059	2,801	8.43%	105	79	24.76%
block_contracts/Simple__square	903	824	8.75%	53	41	22.64%
removeDups/arrayPart	1,809	1,647	8.96%	103	93	9.71%
arith/euclidean/gcdHelp-post	3,135	2,839	9.44%	40	28	30.00%
SparseArray/MemoryAllocator_alloc_unsigned	1,432	1,290	9.92%	90	78	13.33%
java_dl/reverseArray	5,376	4,831	10.14%	151	139	7.95%
java_dl/reverseArray2	2,217	1,992	10.15%	134	109	18.66%
vstte10_04_Queens/Queens_isConsistent	3,729	3,336	10.54%	165	138	16.36%
javacard/arrayFillNonAtomic	5,345	4,773	10.70%	296	276	6.76%
09.list_modelfield/ArrayList.add	2,309	2,060	10.78%	143	132	7.69%
vstte10_01_SumAndMax/SumAndMax_sumAndMax	4,140	3,675	11.23%	140	114	18.57%
java_dl/polishFlagSort	4,482	3,975	11.31%	94	82	12.77%
java_dl/arrayMax	1,951	1,720	11.84%	98	71	27.55%
simple/selection_sort	5,459	4,790	12.25%	277	211	23.83%
09.list_modelfield/ArrayList.remFirst	2,511	2,179	13.22%	208	175	15.87%
comprehensions/segsum	840	719	14.40%	63	52	17.46%
list_seq/ArrayList.contains	2,506	2,141	14.57%	96	62	35.42%
simple/loop2	1,041	883	15.18%	83	58	30.12%
WeideEtAl_01_AddAndMultiply/...add	1,374	1,163	15.36%	109	87	20.18%
simple/oldForParams	558	466	16.49%	48	36	25.00%
SemanticSlicing/project	6,115	5,086	16.83%	436	301	30.96%
removeDups/contains	1,057	876	17.12%	72	53	26.39%
java_dl/java5/for_ReferenceArray	675	555	17.78%	70	45	35.71%
java_dl/java5/for_Array	839	689	17.88%	94	69	26.60%
comprehensions/sum0	798	642	19.55%	84	59	29.76%
arith/cubicSum	1,005	808	19.60%	64	52	18.75%
java_dl/jml-free/loopInvFree	401	322	19.70%	56	45	19.64%
comprehensions/sum1	952	759	20.27%	84	58	30.95%
list_recursiveSpec/...getNextNN	7,199	5,643	21.61%	251	185	26.29%
comprehensions/sum3	844	653	22.63%	99	57	42.42%
comprehensions/sum2	811	625	22.93%	100	58	42.00%
list/ArrayList_contains_dep	6,198	4,364	29.59%	390	218	44.10%
java_dl/java5/for_Iterable	468	314	32.91%	100	57	43.00%
list_seq/ArrayList.remove.0	3,703	2,419	34.67%	186	56	69.89%
fm12_01_LRS/lcp	3,146	1,934	38.53%	235	104	55.74%
block_contracts/...unnecessaryLoopInvariant	105	60	42.86%	27	11	59.26%

Table A.1: Experimental Results of Performance Evaluation without One Step Simplifier, Ordered by the Percentage of Proof Nodes Saved

Problem	Proof Nodes Old	Proof Nodes New	% Diff. # Nodes	# SE Steps Old	# SE Steps New	% Diff. # SE Steps
list/LinkedList_get_normal	4,846	5,341	*-10.21%*	178	166	6.74%
comprehensions/segsum	512	509	0.59%	156	136	12.82%
vstte10_04_Queens/Queens_isConsistent	1,885	1,868	0.90%	165	139	15.76%
SmansEtAl/ArrayList_add	3,838	3,710	3.34%	473	450	4.86%
list/ArrayList_concatenate	9,321	9,010	3.34%	573	505	11.87%
saddleback_search/Saddleback_search	32,337	31,210	3.49%	235	183	22.13%
vstte10_03_LinkedList/Node_search	6,471	6,235	3.65%	97	59	39.18%
WeideEtAl_02_BinarySearch/BinarySearch_search	2,843	2,725	4.15%	182	147	19.23%
list_recursiveSpec/…getNextNN	2,242	2,145	4.33%	216	184	14.81%
vstte10_01_SumAndMax/SumAndMax_sumAndMax	2,438	2,324	4.68%	140	113	19.29%
observer/ExampleSubject_addObserver	2,989	2,835	5.15%	167	135	19.16%
SparseArray/MemoryAllocator_alloc	500	473	5.40%	89	78	12.36%
list_ghost/ArrayList_enlarge	1,535	1,450	5.54%	151	127	15.89%
list_seq/ArrayList.enlarge	1,896	1,780	6.12%	106	81	23.58%
javacard/arrayFillNonAtomic	2,860	2,664	6.85%	296	275	7.09%
list_seq/ArrayList.contains	1,483	1,376	7.22%	96	61	36.46%
list_recursiveSpec/…setValueAt	2,033	1,881	7.48%	189	154	18.52%
list/ArrayList_enlarge	1,794	1,644	8.36%	156	132	15.38%
SparseArray/MemoryAllocator_alloc_unsigned	717	652	9.07%	90	78	13.33%
block_contracts/Simple__square	358	325	9.22%	53	42	20.75%
list_seq/ArrayList.remove.1	8,952	8,031	10.29%	263	195	25.86%
java_dl/reverseArray2	1,125	1,007	10.49%	134	108	19.40%
java_dl/jml-information-flow	43,432	38,867	10.51%	492	435	11.59%
09.list_modelfield/ArrayList.add	1,323	1,182	10.66%	143	131	8.39%
java_dl/arrayMax	1,081	961	11.10%	98	71	27.55%
java_dl/polishFlagSort	3,242	2,869	11.51%	92	80	13.04%
simple/selection_sort	3,813	3,362	11.83%	277	212	23.47%
09.list_modelfield/ArrayList.remFirst	1,530	1,340	12.42%	208	177	14.90%
simple/loop2	594	504	15.15%	83	59	28.92%
arith/cubicSum	502	425	15.34%	64	52	18.75%
WeideEtAl_01_AddAndMultiply/…add	636	537	15.57%	109	86	21.10%
java_dl/java5/for_ReferenceArray	352	295	16.19%	70	46	34.29%
coincidence_count/project	13,819	11,555	16.38%	210	152	27.62%
java_dl/java5/for_Array	392	327	16.58%	95	71	25.26%
java_dl/reverseArray	3,980	3,296	17.19%	128	116	9.38%
comprehensions/sum0	360	294	18.33%	84	58	30.95%
arith/euclidean/gcdHelp-post	2,382	1,940	18.56%	40	28	30.00%
comprehensions/sum1	452	367	18.81%	85	61	28.24%
SemanticSlicing/project	3,469	2,796	19.40%	436	297	31.88%
list/ArrayList_contains_dep	2,950	2,361	19.97%	370	217	41.35%
simple/oldForParams	235	186	20.85%	48	37	22.92%
comprehensions/sum2	397	307	22.67%	100	59	41.00%
comprehensions/sum3	388	300	22.68%	100	60	40.00%
java_dl/jml-free/loopInvFree	189	137	27.51%	56	45	19.64%
java_dl/java5/for_Iterable	230	145	36.96%	100	57	43.00%
removeDups/removeDup	9,475	5,932	37.39%	371	312	15.90%
fm12_01_LRS/lcp	1,812	1,060	41.50%	235	106	54.89%
block_contracts/…unnecessaryLoopInvariant	73	40	45.21%	27	11	59.26%
removeDups/arrayPart	1,741	891	48.82%	101	90	10.89%
list_seq/ArrayList.remove.0	2,992	1,489	50.23%	204	70	65.69%
removeDups/contains	1,040	452	56.54%	72	51	29.17%

Table A.2: Experimental Results of Performance Evaluation with One Step Simplifier, Ordered by the Percentage of Proof Nodes Saved

B Proofs of Abstract Execution Rules

Theorem 4.1. *The AE rule* abstractStatement *(Fig. 4.1) is sound.*

Proof of Thm. 4.1. We have to show (cf. Def. 2.10) that the validity of the sequent in the conclusion of rule abstractStatement follows from the validity of the sequent in the premise. As usual, we show that the *focused formula* in the conclusion is valid, i.e., it holds for an arbitrary structure K and state σ, based on the assumption the premise holds for *all* K' and σ'. Let $\mathcal{F}$ be the abstract program fragment in the conclusion, and P the active AS. According to Def. 4.9, $val(K,\sigma|[\mathcal{F}]\varphi) = tt$ holds if, for all $p \in [\![\mathcal{F}]\!]$, $val(K,\sigma|[p]\varphi) = tt$. We show this fact by case distinction on the completion modes of p, i.e., we distinguish instances that complete normally, complete abruptly because of a thrown exception, because of a return of a value, and so on. We assume that p completes normally iff the corresponding characteristic formula $normalCompletionFor(specs,p)$ holds for p, and similarly for the other completion modes, framing, termination, etc. (cf. Remark 4.6).

Case "p does not terminate". Trivial for the box modality. For diamond, p would not be a legal instantiation of AS P if it did not terminate.

Case "p completes normally". In this case, $val(K,\sigma|[\pi\ p\ \omega]\varphi) = tt$ is equivalent to

$$val(K,\varrho(p)(\sigma)|[\pi\ \omega]\varphi) = tt \tag{B.1}$$

Note that the execution cannot branch, since we evaluate p in a concrete state σ. Because p is a legal instantiation of P, we can derive that $val(K,\sigma|notAbruptly) = ff$ (because, for instance, $pre(excSpec)$ is not satisfiable), `exc` is `null`, and so on. The focused formula in the premise is—specifically for the legal instantiation p—therefore logically equivalent to

$$\begin{array}{l}\{\texttt{throwsExc} := \mathrm{FALSE} \parallel \ldots\}\texttt{normal} \doteq \mathrm{TRUE} \rightarrow \\ \quad \{\mathcal{U}_{\mathsf{P}}(frame :\approx value(footprint))\}\{\texttt{exc} := exc^{\mathsf{P}}(value(footprint)) \parallel \ldots\} \\ \qquad \big(post(normalSpec) \rightarrow [\pi\ \texttt{if (throwsExc) throw exc; ...}\ \omega]\varphi\big)\end{array}$$

$$
\text{abstractStatement}\quad
\frac{\begin{array}{l}
\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{init}\} \\
\quad \big(\mathit{mutex}(\mathtt{throwsExc}, \mathtt{returnsVal}, \mathtt{returns}, \mathtt{breaks}, \mathtt{continues}, \\
\qquad \mathtt{breaks\_}\mathit{lb}_{b_1}, \ldots, \mathtt{breaks\_}\mathit{lb}_{b_n}, \\
\qquad \mathtt{continues\_}\mathit{lb}_{c_1}, \ldots, \mathtt{continues\_}\mathit{lb}_{c_m}) \land \\
\quad (\mathtt{normal} \doteq \mathrm{TRUE} \leftrightarrow \mathit{notAbruptly}) \land \\
\quad \mathit{behavioralPreconds}\big) \rightarrow \\
\quad \{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\} \\
\quad \{\mathtt{exc} := \mathit{exc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint})) \mid\mid \mathtt{res} := \mathit{res}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\} \\
\quad \big(\mathit{behavioralPostconds} \rightarrow \\
\qquad [\pi\ \ \mathbf{if}\ \mathtt{(throwsExc)}\ \mathbf{throw}\ \mathtt{exc;} \\
\qquad\quad \mathbf{if}\ \mathtt{(returnsVal)}\ \mathbf{return}\ \mathtt{res;} \\
\qquad\quad \mathbf{if}\ \mathtt{(returns)}\ \mathbf{return}\mathtt{;} \\
\qquad\quad \mathbf{if}\ \mathtt{(breaks)}\ \mathbf{break}\mathtt{;} \\
\qquad\quad \mathbf{if}\ \mathtt{(continue)}\ \mathbf{continue}\mathtt{;} \\
\qquad\quad \mathbf{if}\ (\mathtt{breaks\_}\mathit{lb}_{b_1})\ \mathbf{break}\ \mathit{lb}_{b_1}\mathtt{;}\ \cdots \\
\qquad\quad \mathbf{if}\ (\mathtt{breaks\_}\mathit{lb}_{b_n})\ \mathbf{break}\ \mathit{lb}_{b_n}\mathtt{;} \\
\qquad\quad \mathbf{if}\ (\mathtt{continues\_}\mathit{lb}_{c_1})\ \mathbf{continue}\ \mathit{lb}_{c_1}\mathtt{;}\ \cdots \\
\qquad\quad \mathbf{if}\ (\mathtt{continues\_}\mathit{lb}_{c_m})\ \mathbf{continue}\ \mathit{lb}_{c_m}\mathtt{;}\ \ \omega]\varphi\big), \Delta
\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \mathbf{\backslash abstract\_statement}\ \mathtt{P;}\ \ \omega]\varphi, \Delta}
$$

Figure B.1: The Abstract Execution Rule for Abstract Statements. (Abbreviations and label symbols are explained around Page 151)

Since the variables `throwsExc`, `exc` etc. are fresh (i.e., they do not occur in φ, *footprint*, ...), we can substitute their occurrences by the right-hand sides in the updates and remove the latter. Furthermore, *post*(*normalSpec*) can be assumed to be logically valid (again, because p is a legal instantiation). The above formula is thus logically equivalent to

$$\{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\}([\pi\ \omega]\varphi) \tag{B.2}$$

Remark. Since the abstract update is not created fresh, but *dependently fresh* for P, i.e., is re-used if P occurred before in the proof, we *cannot assume the validity* of Eq. (B.2) at this point. Re-using an identifier in second-order Skolemization, to which abstract updates essentially amount, is a non-trivial operation. Compare this to standard Skolemization of universally quantified variables: If the constant introduced for the variable was *not* fresh, the validity of the premise would not imply the validity of the quantified formula for *all* domain elements, but only for those meeting the constraints expressed in the context. For the case of abstract updates, an already existing premise $\{\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{x}}^! :\approx \mathit{args})\}\mathsf{x} \doteq 17$ in the context, for example, would restrict the interpretations of the symbol $\mathcal{U}_{\mathsf{P}}(\mathring{\mathsf{x}}^!)$ to those that, when applied with arguments *args*, assign to x the value 17—even though we cannot *directly* use abstract updates in (in)equations, but only apply them to terms and formulas. The reasoning behind the admissibility of re-using the abstract update symbol is that it is not an arbitrary symbol from the proof context, but created *dependently* fresh *for the AS* P. It would indeed be a problem if this symbol was used in a manner that was not justified. However, since it is created *fresh* upon first appearance of P and is *only* re-used for APEs *of the same identifier symbol* which are guaranteed to be *behaviorally isomorphic* by the definition of their semantics, relevant interpretations, like the one that we choose subsequently, will not be excluded. Finally, it is important that abstract updates have state-dependent parameters *value*(*footprint*), which ensures that abstract updates in different contexts can still be instantiated differently. It would be unsound to *re*-use abstract updates, or first-order logic symbols (cf. the case of completion due to a thrown exception below), *without* parameters. $\Diamond$

Let $\mathcal{U}_p$ be an update which is logically equivalent to $\varrho(p)$. Since p is a legal instantiation of P, *frame* and *footprint* are valid (abstract) upper bounds on the locations that are written / read by p. Consequently, and considering the remark above, there has to be an interpretation of the form $\mathcal{U}_p \,||\, \mathcal{U}'$ of the abstract update, where $\mathcal{U}'$ is an abstract update with left-hand sides that are disjoint from those in $\mathcal{U}_p$. Note that in $\mathcal{U}'$, there can be no **\hasTo** locations, since otherwise, they had to be also in $\mathcal{U}_p$. Therefore, there is a possible interpretation of $\mathcal{U}'$ as *Skip*. Since $\mathcal{U}_p$ is a concrete update, we can assume the validity of the instantiation: $\models \{\mathcal{U}_p\}([\pi\ \omega]\varphi)$. We specialize this to $\mathit{val}\big(K, \sigma | \{\mathcal{U}_p\}([\pi\ \omega]\varphi)\big) = \mathit{tt}$, which in turn is equivalent to $val(K, \varrho(p)(\sigma)|[\pi\ \omega]\varphi) = tt$.

Case "p completes due to a thrown exception". Let exc_0 be the thrown exception; $val(K, \sigma|[\pi\ p\ \omega]\varphi) = tt$ is then equivalent to

$$val\big(K, \varrho(p')(\sigma)|[\pi\ \texttt{throw}\ exc_0\texttt{;}\ \ \omega]\varphi\big) = tt \tag{B.3}$$

where p' is the prefix of p that is executed before the exception is thrown (including the content of **`finally`** and **`catch`** blocks in the order of execution). Since we assume $excFor(excSpec, p)$ to be valid, the formula $pre(excSpec)$ has to be valid, and all other behavioral preconditions are unsatisfiable (mutual exclusion follows from Def. 4.1, validity of $pre(excSpec)$ follows from the fact that normal completion can be excluded, i.e., $normalCompletionFor(specs, p)$ is be unsatisfiable, which implies that the formula *normal* is unsatisfiable, which in turn implies, together with the assumption of the validity of $excFor(excSpec, p)$, that $pre(excSpec)$ is valid). Therefore, the focused formula in the premise can, again preserving validity, be simplified to

$$\begin{aligned}&\{\mathcal{U}_{\mathsf{P}}(frame :\approx value(footprint))\}\\ &\qquad\{\texttt{exc} := exc^{\mathsf{P}}(value(footprint))\}\big(post(excSpec) \wedge \texttt{exc} \neq \texttt{null} \rightarrow\\ &\qquad\qquad\qquad\qquad\qquad\qquad [\pi\ \texttt{throw exc;}\ \ \omega]\varphi\big)\end{aligned}$$

The significant difference to the normal completion case is the presence of the thrown exception object. If `exc` was initialized to a fresh Skolem constant, it would be clear that there is a model interpreting it with the actually thrown exc_0; after also suitably instantiating the abstract update, we could close this case. However, `exc` is initialized to the term $exc^{\mathsf{P}}(value(footprint))$, where the symbol exc^{P} is created *dependently* fresh for P. Note that also the postcondition $post(excSpec)$ can constrain the value of `exc`, which is, however, no problem, since we can assume p to respect the specification, which implies that this does *not* rule out exc_0 from the possible instantiations. As for abstract updates, it is admissible to re-use this the symbol exc_{P} because it is created fresh on the first occurrence of an AS with identifier P, and *only* re-used by the AE rules for occurrences of ASs with the same identifier. These ASs can, however, be expected to throw the *same* exception object when invoked in the same pre-state. Thus, exc_0 is a valid instantiation, since any ASs occurred before has a different identifier symbol, was executed in a different state, *or* threw the same exception. We thus specialize the premise to

$$\{\mathcal{U}_{\mathsf{P}}(frame :\approx value(footprint))[\pi \ \texttt{throw}\ exc_0\texttt{;}\ \ \omega]\varphi$$

from where we can continue as in the normal behavior case, exploiting the fact that *frame* and *footprint* are *upper bounds* of the actually assigned and accessed locations and we can therefore instantiate them to reflect the effects of the program prefix p'.

Remaining abrupt completion cases. The remaining cases work analogously; for the case "completion due to a **`return`** of a value", the same considerations apply for the returned value as for the thrown exception in the exceptional completion case. The other cases, where this part does not apply, are otherwise identical. □

Remark B.1 (Dependently Fresh Creation of Symbols). Considering that the usage of (non-dependent) *fresh* Skolem constants (or abstract updates) would render the soundness argument much less complicated, one might wonder why we do not simply use these instead. The immediate, formalistic reason is the adherence to principle (2) for the design of AE rules. The actual motivation, also for the principle, is that this ensures the *completeness* of the rules. Two instances of the same APE can be expected to throw the same exception object when invoked in the same state, and generally have the same effects on the post-state when invoked in the same pre-state. With pure Skolemization, this could not be achieved without a non-trivial postcondition. For thrown exceptions and returned values, it would be sufficient manually bind them to suitable expressions (such as *excObjP*(*value*(*footprint*)), where *excObjP* is a user-defined function symbol). To compensate for the lack of dependently fresh abstract *updates*, it would generally be necessary to specify the whole relevant part of the post-state to re-establish completeness in the presence of more than one APE with the same identifier symbol (which is roughly comparable to loop invariant rules masking the complete context). ◊

Theorem 4.3. *The AE rule* abstractStatement *(Fig. 4.1) is complete.*

Proof of Thm. 4.3. To prove completeness of abstractStatement, we have to show that the validity of the premise of the rule follows from the validity of the conclusion (the reverse direction of Thm. 4.1, cf. Def. 2.10). We assume that all the premises of the implication cascade in the rule's premise evaluate to *tt*; otherwise, the sequent is trivially valid. Note that the *mutex* premise and the constraint `normal` $\leftrightarrow$ *notAbruptly* ensure that *exactly* one of those evaluates to *tt*. We can therefore, as in the soundness case, proceed by case distinction, in this case on the value of the flags `normal`, `throwsExc` etc. Let K, σ be an arbitrary structure and state; we show that the premise is valid for those.

Case "`normal` evaluates to *tt*". In this case, due to mutual exclusion of the premises for the different completion types, the focused formula in the premise collapses to

$$\{\mathscr{U}_{\mathsf{P}}(frame :\approx value(footprint))\}([\pi \quad \omega]\varphi)$$

which we have to show true. We obtain, where I is the interpretation function of K,

$$\begin{aligned} & val(K, \sigma|\{\mathscr{U}_{\mathsf{P}}(frame :\approx value(footprint))\}([\pi \quad \omega]\varphi)) \doteq tt \\ \iff & val(K, I(\mathscr{U}_{\mathsf{P}}(frame))(value(footprint))(\sigma)|([\pi \quad \omega]\varphi)) \doteq tt \end{aligned}$$

Note that $I(\mathscr{U}_{\mathsf{P}}(frame))$ can be constrained by already existing formulas in the context Γ, Δ—which for the completeness case does not complicate the proof by requiring more justification, but makes it *simpler*, as we do not have to prove validity for all, but for possibly *less* interpretations of the abstract updates symbol.

Let p' be a normally completing legal instantiation of AS P. Such an instantiation has to exist, since otherwise, `normal` could not evaluate to *tt* due to the definition of the semantics of APEs. Therefore, there have to be structures K' and states σ' s.t.

$$val\big(K', \varrho(p')(\sigma')|[\pi \quad \omega]\varphi\big) = tt$$

If there is a specific, normally completing legal instantiation p for which

$$val(K, \varrho(p)(\sigma)|[\pi \quad \omega]\varphi) = tt$$

holds and furthermore we have that

$$I(\mathscr{U}_{\mathsf{P}}(frame))(value(footprint)) = \varrho(p),$$

i.e., p transforms states the same way that the interpretation of the abstract update symbol by I does, we can close the proof.

The tricky part of the completeness proof, where the *dependent* introduction of fresh symbols comes into play, is that due to the definition of the semantics of abstract sequents and program fragments (Defs. 4.5 and 4.9), instances of APEs with the same identifier symbols are behaviorally isomorphic. This restricts the possible valuations of $\varrho(p)$. If $\mathscr{U}_{\mathsf{P}}$ was introduced fresh "in isolation", there would be valuations of the abstract update that could not be attained by the instantiations of the AS. At this place, we employ a non-standard argument: The *interpretations* of the abstract update are not coupled to those of the APEs with the same identifier; however, we know that there is only one calculus rule for ASs (and one for AExps), and they will always re-use the same (initially fresh) abstract update symbols. Thus, also abstract updates introduced for APEs with the same identifier symbols are coupled (since their interpretations are isomorphic, cf. Def. 4.7). Exploiting this implicit coupling inducing a restriction of the interpretations $I(\mathcal{U}_{\mathsf{P}}(\mathit{frame}))(\mathit{value}(\mathit{footprint}))$, and that legal instantiations of ASs, as well interpretations of the abstract update, both respect *frame* and *footprint*, we conclude that there is a suitable p for all ultimately relevant interpretations of the abstract update.

Case “`throwsExc` evaluates to *tt*”. In this case, the focused formulas simplifies to

$$\begin{aligned}&\{\mathcal{U}_{\mathsf{P}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint}))\}\\ &\qquad\{\mathtt{exc} := \mathit{exc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\}\big(\mathit{post}(\mathit{excSpec}) \land \mathtt{exc} \not\doteq \mathtt{null} \rightarrow\\ &\qquad\qquad\qquad\qquad\qquad\qquad [\pi\ \mathbf{throw}\ \mathtt{exc};\ \omega]\varphi\big)\end{aligned}$$

and we have to show, for $\sigma' = I(\mathcal{U}_{\mathsf{P}}(\mathit{frame}))(\mathit{value}(\mathit{footprint}))(\sigma)$, that

$$\mathit{val}\big(K, \sigma' | ([\pi\ \mathbf{throw}\ \mathtt{exc};\ \omega]\varphi)\big) \doteq \mathit{tt}$$

where

$$\begin{aligned}&\mathit{val}\big(K, \sigma' | \mathtt{exc}\big) = \mathit{val}\big(K, \sigma' | \mathit{exc}^{\mathsf{P}}(\mathit{value}(\mathit{footprint}))\big) \text{ and}\\ &\mathit{val}\big(K, \sigma' | \big(\mathit{post}(\mathit{excSpec}) \land \mathtt{exc} \not\doteq \mathtt{null}\big) = \mathit{tt}.\end{aligned}$$

Let p'' be an AS which completes due to a thrown exception exc_0. Analogously to the normal completion case, such a statement has to exist, since $\mathit{pre}(\mathit{excSpec})$ is assumed to evaluate to *tt*. Let furthermore p' be the prefix of p'' that is executed before the exception is thrown (including the content of **`finally`** and **`catch`** blocks in the order of execution). Therefore, there have to exist structures K' and states σ'' such that

$$\mathit{val}\big(K', \varrho(p')(\sigma') | [\pi\ \mathbf{throw}\ \mathit{exc}_0;\ \omega]\varphi\big) = \mathit{tt},$$

Accordingly, we have to find a specific legal instantiation with prefix p s.t.

$$\mathit{val}(K, \varrho(p)(\sigma) | [\pi\ \mathbf{throw}\ \mathit{exc}_0;\ \omega]\varphi) = \mathit{tt} \quad \text{and} \tag{B.4}$$

$$\sigma' = \varrho(p) \quad \text{and} \tag{B.5}$$

$$\mathit{val}\big(K, \sigma' | \mathtt{exc}\big) = \mathit{exc}_0 \tag{B.6}$$

Equation (B.4) is satisfied if Eq. (B.5) holds, since we can assume that in this state, an exception is thrown. The argument for Eq. (B.5) is as for normal completion. For the case of Eq. (B.6), the thinking is similar: APEs with the same identifier symbol throw the same exception object when called in the same state. Since we also choose the symbol exc_{P} dependently fresh, this rules out potential interpretations that are excluded by the requirement of isomorphic behavior. Note that the requirement on *post*(*excSpec*) is unproblematic, since legal instantiations can be assumed to adhere to the specifications. Also, there has to be a non-null exc_0, since throwing `null` amounts to throwing a new `NullPointerException`, which will be the ultimate "effect" of the exceptionally completing legal instantiation.

Remaining cases. The remaining cases work analogously. □

C Abstract Execution Taclets

Below, we show the taclet for symbolic execution of an AS within the context of a loop and a non-void method. This taclet exactly corresponds to the implementation, and deviates in some aspects from the rules in text-book style shown in the thesis. First, it is only applicable in the mentioned context; there are other rules for occurrences of ASs outside of a loop, and for void methods. Therefore, there is also no distinction of an abrupt completion due to a return and a return of a value, since only one of those (the latter) is possible in that context. Second, there is so far no support for the specification of pre- and postconditions for labeled **break**s and **continue**s. Instead, the rule will create unconstrained SE branches for each label occurring in the context. The taclet extends the text-book rule by an additional assumption related to the frame condition (lines 62 to 66). The taclet for AExps is displayed afterward.

Listing C.1: Taclet for Abstract Statements in a Loop Context

```
1 abstractStatement {
2     \schemaVar \update U;
3
4     \find (\modality{#allmodal}{ .. #absProg ... }\endmodality(post))
5
6     \varcond(\prefixContainsElement("LoopScopeBlock"))
7     \varcond(\prefixContainsElement("MethodFrame"))
8
9     \varcond(\storeResultVarIn(#v))
10    \varcond(\isDefined(#v))
11
12    \varcond(\storeContextLabelsIn(#labels))
13    \varcond(\storeContextLoopLabelsIn(#labels1))
14    \varcond(\instantiateVarsFresh(
15      #vars, #labels, "breaks", boolean \freshFor(#absProg)))
16    \varcond(\instantiateVarsFresh(
17      #vars1, #labels1, "continues", boolean \freshFor(#absProg)))
18
```

```
    \varcond(\new(#normal, boolean))
    \varcond(\new(#throwsExc, boolean))
    \varcond(\new(#exc, java.lang.Throwable))
    \varcond(\new(#returns, boolean))
    \varcond(\new(#result, \typeof(#v)))
    \varcond(\new(#breaks, boolean))
    \varcond(\new(#continues, boolean))
    \varcond(\new(#h, Heap))

    \varcond(\initializeParametricSkolemUpdate(U, #absProg))

    // Index variables for foreach loop
    \varcond(\new(#v1, boolean))
    \varcond(\newLabel(#label))

    \replacewith (
        {   #normal:=#abstrPrecond(#absProg, "normal")
         || #throwsExc:=#abstrPrecond(#absProg, "throwsExc")
         || #returns:=#abstrPrecond(#absProg, "returns")
         || #breaks:=#abstrPrecond(#absProg, "breaks")
         || #continues:=#abstrPrecond(#absProg, "continues")
         || #h:=heap
        }
         (
           (  #mutualExclusionFormula5(
                #returns, #throwsExc, #breaks, #continues, #vars)
            & (#normal = TRUE <-> !#returns = TRUE &
                 !#throwsExc = TRUE & !#breaks = TRUE & !#continues = TRUE)
            & #returnPrecondition(#absProg, #returns)
            & #excPrecondition(#absProg, #throwsExc)
            & #breaksPrecondition(#absProg, #breaks)
            & #continuesPrecondition(#absProg, #continues)
           ) ->
           {U}{#exc:=#abstrPrecond(#absProg, "exceptionObject") ||
               #result:=#addCast(
                 #abstrPrecond(#absProg, "resultObject"), #result)}
             (
               ((#returns = TRUE ->
                   #postCondAE(#absProg, "returns", #returns, #result, #exc)) &
                (#throwsExc = TRUE -> !#exc = null &
                   #postCondAE(#absProg, "throwsExc", #returns, #result, #exc)) &
                (#normal = TRUE ->
```

```
                    #postCondAE(#absProg, "normal", #returns, #result, #exc)) &
                  (\forall f; \forall o;
                    (  elementOf(o,f,#getFrame(#absProg))
                     | !o=null &
                       !boolean::select(#h,o,java.lang.Object::<created>)=TRUE
                     | any::select(heap,o,f) = any::select(#h,o,f)))) ->
                    \modality{#allmodal}{
                      ..
                      if (#returns) {
                          return #result;
                      }
                      if (#throwsExc) {
                          throw #exc;
                      }
                      if (#continues) {
                          continue;
                      }
                      if (#breaks) {
                          break;
                      }
                      #foreach (#v1, #label in #vars, #labels) {
                          if (#v1) {
                              break #label;
                          }
                      }
                      #foreach (#v1, #label1 in #vars1, #labels1) {
                          if (#v1) {
                              continue #label1;
                          }
                      }
                      ...
                    }\endmodality(post)
              )
          )
    )

    \heuristics(abstractExecution, simplify_prog)
};
```

The taclet for AExps is shown subsequently. This taclet, which is implemented exactly as shown, realizes all the aspects of the text-book-style rule abstractExpression which we introduced in Sect. 4.3. Similarly to the taclet for ASs, it extends the text-book rule by the assertion on the frame in lines 30 to 34.

```
1  abstractExpression {
2      \schemaVar \update U;
3
4      \find (\modality{#allmodal}{ .. #v = #aexp; ... }\endmodality(post))
5
6      \varcond(\new(#normal, boolean))
7      \varcond(\new(#throwsExc, boolean))
8      \varcond(\new(#exc, java.lang.Throwable))
9      \varcond(\new(#returns, boolean))
10     \varcond(\new(#result, \typeof(#v)))
11     \varcond(\new(#h, Heap))
12
13     \varcond(\initializeParametricSkolemUpdate(U, #aexp))
14
15     \replacewith (
16          {   #normal:=#abstrPrecond(#aexp, "normal")
17           || #throwsExc:=#abstrPrecond(#aexp, "throwsExc")
18           || #h:=heap}
19           (
20             (   (#normal = TRUE <-> !#throwsExc = TRUE)
21              & (#excPrecondition(#aexp, #throwsExc))
22              & (#throwsExc = TRUE -> !#exc = null)) ->
23             {U}{#exc:=#abstrPrecond(#aexp, "exceptionObject") ||
24                 #result:=#addCast(#abstrPrecond(#aexp, "resultObject"), #v)}
25               (
26                 (( #throwsExc = TRUE -> !#exc = null &
27                      #postCondAE(#aexp, "throwsExc", #returns, #result, #exc)) &
28                  (!#throwsExc = TRUE ->
29                      #postCondAE(#aexp, "normal", #returns, #result, #exc)) &
30                  (\forall f; \forall o;
31                    (  elementOf(o,f,#getFrame(#aexp))
32                     | !o=null &
33                       !boolean::select(#h,o,java.lang.Object::<created>)=TRUE
34                     | any::select(heap,o,f) = any::select(#h,o,f)))) ->
35                    \modality{#allmodal}{
36                      ..
37                      if (#throwsExc) {
38                          throw #exc;
39                      }
40
41                      #v = #result;
```

```
                        ...
                      }\endmodality(post)
                )
            )
    )

    \heuristics(abstractExecution, simplify_prog)
};
```

D MTL and STL Proofs

This section contains the proof demonstrating which axioms of PDL are satisfied by MTL, as well as the soundness proof of the STL calculus.

Proof of Lem. 5.4. Let β, γ, φ and ψ have a trace semantics, and α be a trace abstraction.

Axiom (iii): $[\beta \Vdash_\alpha \varphi \odot \psi] \equiv [\beta \Vdash_\alpha \varphi] \odot [\beta \Vdash_\alpha \psi]$. ✓/✗

$$\begin{aligned}
& tval(K,\sigma|[\beta \Vdash_\alpha \varphi \odot \psi]) \\
=\ & \overline{\alpha(tval(K,\sigma|\beta))} \cup \alpha(tval(K,\sigma|\varphi \odot \psi)) \\
=\ & \overline{\alpha(tval(K,\sigma|\beta))} \cup (\alpha(tval(K,\sigma|\varphi) \cap tval(K,\sigma|\psi))) \\
\neq\ & \overline{\alpha(tval(K,\sigma|\beta))} \cup (\alpha(tval(K,\sigma|\varphi)) \cap \alpha(tval(K,\sigma|\psi))) \\
=\ & \left(\overline{\alpha(tval(K,\sigma|\beta))} \cup \alpha(tval(K,\sigma|\varphi))\right) \cap \left(\overline{\alpha(tval(K,\sigma|\beta))} \cup \alpha(tval(K,\sigma|\psi))\right) \\
=\ & [\beta \Vdash_\alpha \varphi] \odot [\beta \Vdash_\alpha \psi]
\end{aligned}$$

This axiom only holds for abstractions which are homomorphic for intersections, e.g., the identity abstraction.

Axiom (iv): $[\beta \cup \gamma \Vdash_\alpha \varphi] \equiv [\beta \Vdash_\alpha \varphi] \odot [\gamma \Vdash_\alpha \varphi]$. ✓

$$\begin{aligned}
& tval(K,\sigma|[\beta \cup \gamma \Vdash_\alpha \varphi]) \\
=\ & \overline{\alpha(tval(K,\sigma|\beta \cup \gamma))} \cup \alpha(tval(K,\sigma|\varphi)) \\
=\ & \overline{\alpha(tval(K,\sigma|\beta) \cup \alpha(tval(K,\sigma|\gamma)))} \cup \alpha(tval(K,\sigma|\varphi)) \\
=\ & \overline{\alpha(tval(K,\sigma|\beta)) \cup \alpha(tval(K,\sigma|\gamma))} \cup \alpha(tval(K,\sigma|\varphi)) \\
=\ & \left(\overline{\alpha(tval(K,\sigma|\beta))} \cap \overline{\alpha(tval(K,\sigma|\gamma))}\right) \cup \alpha(tval(K,\sigma|\varphi)) \\
=\ & \left(\overline{\alpha(tval(K,\sigma|\beta))} \cup \alpha(tval(K,\sigma|\varphi))\right) \cap \left(\overline{\alpha(tval(K,\sigma|\gamma))} \cup \alpha(tval(K,\sigma|\varphi))\right) \\
=\ & tval(K,\sigma|[\beta \Vdash_\alpha \varphi]) \odot tval(K,\sigma|[\gamma \Vdash_\alpha \varphi])
\end{aligned}$$

Axiom (v): $[\beta;\gamma \Vdash_\alpha \varphi] \equiv [\beta \Vdash_\alpha [\gamma \Vdash_\alpha \varphi]]$. ✗

$$\begin{aligned}
& tval(K,\sigma|[\beta;\gamma \Vdash_\alpha \varphi]) \\
={}& \overline{\alpha(tval(K,\sigma|\beta;\gamma))} \cup \alpha(tval(K,\sigma|\varphi)) \\
\neq{}& \overline{\alpha(tval(K,\sigma|\beta)) \cap \alpha(tval(K,\sigma|\gamma))} \cup \alpha(tval(K,\sigma|\varphi)) \\
={}& \overline{\alpha(tval(K,\sigma|\beta))} \cup \overline{\alpha(tval(K,\sigma|\gamma))} \cup \alpha(tval(K,\sigma|\varphi))
\end{aligned}$$

The equality $\alpha(tval(K,\sigma|\beta;\gamma)) = \alpha(tval(K,\sigma|\beta)) \cap \alpha(tval(K,\sigma|\gamma))$ only holds for special cases, e.g., big-step abstraction and constructs β, γ which do not perform conflicting changes on the state. Thus, the linearization axiom (v) does generally not hold.

Axiom (vi): $[\psi? \Vdash_\alpha \varphi] \equiv (\psi \subset \varphi)$. ✗

$$\begin{aligned}
& [\psi? \Vdash_\alpha \varphi] \\
={}& \overline{\alpha(tval(K,\sigma|\psi?))} \cup \alpha(tval(K,\sigma|\varphi)) \\
\neq{}& \overline{tval(K,\sigma|\psi)} \cup tval(K,\sigma|\varphi) \\
={}& \psi \subset \varphi
\end{aligned}$$

This axiom only holds if ψ is *unsatisfiable*. Otherwise, the set $tval(K,\sigma|\psi)$ will almost always be different than the (usually much smaller) $\alpha(tval(K,\sigma|\psi?))$, and the traces of φ are abstracted in the left- but not the right-hand side.

Axioms (vii) and (viii): Unwinding with linearization and induction axiom. ✗

These axioms do also not hold since they rely on linearization, which does not work for the trace modality (see axiom (v)). □

Proof of Thm. 5.11. +left:

$$\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\varpi) \cup tval\big(K,\sigma|\varpi'\big) \cup \overline{\bigcap tval(K,\sigma|\Delta)} \stackrel{\text{def.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi+\varpi'\big) \cup \overline{\bigcap tval(K,\sigma|\Delta)}.$$

+right:

$$\Big(\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi) \cap \bigcap tval(K,\sigma|\Delta)}\Big) \cap \Big(\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi') \cap \bigcap tval(K,\sigma|\Delta)}\Big)$$

$$
\begin{aligned}
&\overset{\text{de Morgan}}{=} \Big(\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi)} \cup \overline{\bigcap tval(K,\sigma|\Delta)}\Big) \cap \\
&\qquad \Big(\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi)'} \cup \overline{\bigcap tval(K,\sigma|\Delta)}\Big) \\
&\overset{\text{dist.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup \Big(\overline{tval(K,\sigma|\varpi)} \cap \overline{tval(K,\sigma|\varpi')}\Big) \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
&\overset{\text{invol.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup \overline{\overline{\Big(\overline{tval(K,\sigma|\varpi)} \cap \overline{tval(K,\sigma|\varpi')}\Big) \cup \overline{\bigcap tval(K,\sigma|\Delta)}}} \\
&\overset{\text{de Morgan}}{=} \bigcup tval(K,\sigma|\Gamma) \cup \overline{(tval(K,\sigma|\varpi) \cup tval(K,\sigma|\varpi')) \cap \bigcap tval(K,\sigma|\Delta)} \\
&\overset{\text{def.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi + \varpi') \cap \bigcap tval(K,\sigma|\Delta)}
\end{aligned}
$$

$\neg$left:

$$
\begin{aligned}
&\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi) \cap \bigcap tval(K,\sigma|\Delta)} \\
\overset{\text{de Morgan}}{=} &\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi)} \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
\overset{\text{def.}}{=} &\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\neg\varpi) \cup \overline{\bigcap tval(K,\sigma|\Delta)}
\end{aligned}
$$

$\neg$right:

$$
\begin{aligned}
&\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\varpi) \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
\overset{\text{de Morgan}}{=} &\bigcup tval(K,\sigma|\Gamma) \cup \overline{\overline{tval(K,\sigma|\varpi)} \cap \bigcap tval(K,\sigma|\Delta)} \\
\overset{\text{def.}}{=} &\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\neg\varpi) \cap \bigcap tval(K,\sigma|\Delta)}
\end{aligned}
$$

elim$\top^\infty$:

$$
\begin{aligned}
&\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\top^\infty;\varpi) \cup tval\big(K,\sigma|\varpi';\varpi\big) \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
\overset{\text{def.}}{=} &\bigcup tval(K,\sigma|\Gamma) \cup \{\tau \in Traces \mid \neg finite(\tau)\} \cup \big\{\tau\tau' \mid finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\big\} \cup \\
&\quad \big\{\tau \in tval\big(K,\sigma|\varpi'\big) \mid \neg finite(\tau)\big\} \cup \\
&\quad \big\{\tau\tau' \mid \tau \in tval\big(K,\sigma|\varpi'\big) \wedge finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\big\} \cup \\
&\quad \overline{\bigcap tval(K,\sigma|\Delta)}
\end{aligned}
$$

$$\begin{aligned}
&\stackrel{(\star)}{=} \bigcup tval(K,\sigma|\Gamma) \cup \{\tau \in Traces \mid \neg finite(\tau)\} \cup \left\{\tau\tau' \mid finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\right\} \cup \\
&\qquad \overline{\bigcap tval(K,\sigma|\Delta)} \\
&\stackrel{\text{def.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\top^\infty;\varpi) \cup \overline{\bigcap tval(K,\sigma|\Delta)}
\end{aligned}$$

where $(\star)$ means elimination of subsets in unions, i.e., $A \subseteq B$ iff $B \cup A = B$.

elim$\top$:

$$\begin{aligned}
&\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\top;\varpi) \cup tval\big(K,\sigma|\varpi';\varpi\big) \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
&\stackrel{\text{def.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup \left\{\tau\tau' \mid finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\right\} \cup \\
&\qquad \left\{\tau \in tval\big(K,\sigma|\varpi'\big) \mid \neg finite(\tau)\right\} \cup \\
&\qquad \left\{\tau\tau' \mid \tau \in tval\big(K,\sigma|\varpi'\big) \wedge finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\right\} \cup \\
&\qquad \overline{\bigcap tval(K,\sigma|\Delta)} \\
&\stackrel{(\dagger)}{=} \bigcup tval(K,\sigma|\Gamma) \cup \left\{\tau\tau' \mid finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\right\} \cup \\
&\qquad \left\{\tau\tau' \mid \tau \in tval\big(K,\sigma|\varpi'\big) \wedge finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\right\} \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
&\stackrel{(\star)}{=} \bigcup tval(K,\sigma|\Gamma) \cup \left\{\tau\tau' \mid finite(\tau) \wedge \tau' \in tval(K,\sigma|\varpi)\right\} \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
&\stackrel{\text{def.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\top;\varpi) \cup \overline{\bigcap tval(K,\sigma|\Delta)}
\end{aligned}$$

where $(\dagger)$ means that ϖ' cannot have infinite traces, since per Def. 5.7, only symbolic traces with occurrences of $\top^\infty$ or $(\varpi'')^\omega$ (for any ϖ'') represent infinite traces, which is excluded; and $(\star)$ is as for elim$\top^\infty$.

elim;: This proof does not work by Lem. 5.10, since the rule is *incomplete*: Since we discard the context, one or both premises can be invalid, even if the conclusion *was* valid. For soundness, we have to prove that from the universal validity of the premises, the universal validity of the conclusion follows. The latter is phrased as, for any K, σ,

$$\bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi;\varpi''\big) \cup \overline{tval(K,\sigma|\varpi';\varpi''') \cap \bigcap tval(K,\sigma|\Delta)} = Traces.$$

I.e., by Def. 5.7, we have to show

$$\begin{aligned}
&\bigcup tval(K,\sigma|\Gamma) \cup \{\tau \in tval(K,\sigma|\varpi) \mid \neg finite(\tau)\}\\
&\qquad \cup \left\{\tau\tau' \mid \tau \in tval(K,\sigma|\varpi) \wedge finite(\tau) \wedge \tau' \in tval\left(K,\sigma|\varpi''\right)\right\} \supseteq\\
&\qquad\qquad \left(\left\{\tau \in tval\left(K,\sigma|\varpi'\right) \mid \neg finite(\tau)\right\}\right.\\
&\qquad\qquad \left.\cup \left\{\tau\tau' \mid \tau \in tval\left(K,\sigma|\varpi'\right) \wedge finite(\tau) \wedge \tau' \in tval\left(K,\sigma|\varpi'''\right)\right\}\right)
\end{aligned}$$

where we may assume that

$$tval(K,\sigma|\varpi) \supseteq tval\left(K,\sigma|\varpi'\right) \tag{D.1}$$

$$tval\left(K,\sigma|\varpi''\right) \supseteq tval\left(K,\sigma|\varpi'''\right) \tag{D.2}$$

Consider a trace of Δ which is also either an infinite trace of ϖ' or a trace starting with a finite trace of ϖ' and ending with a trace of ϖ'''. In the first case, by Eq. (D.1) it is also an infinite trace of ϖ, so the subset relation holds. If it is a trace $\tau\tau'$, where τ is a finite trace of ϖ' and τ' a trace of ϖ''', then by Eq. (D.1) it is a trace of $\varpi;\varpi'''$. Moreover, by Eq. (D.2), every trace of ϖ''' is also a trace of ϖ'', therefore, $\tau\tau'$ is a trace of $\varpi;\varpi''$.

If the context Γ, Δ was not removed, Eqs. (D.1) and (D.2) would not be useful; each trace of, e.g., ϖ''' could then also be a trace of Γ, which would neither imply that $\tau\tau'$ is a trace of Γ nor a trace of $\varpi;\varpi''$. Therefore elim; had to be designed incomplete.

elim$^*{}_1$:

$$\begin{aligned}
&\bigcup tval(K,\sigma|\Gamma) \cup tval\left(K,\sigma|\varpi';\varpi^*;\varpi''\right) \cup tval\left(K,\sigma|\varpi';\varpi''\right) \cup \overline{\bigcap tval(K,\sigma|\Delta)}\\
\overset{\text{def.}}{=}\ &\bigcup tval(K,\sigma|\Gamma) \cup \left\{\tau'\tau_1\cdots\tau_n\tau'' \mid \tau_i \in tval(K,\sigma|\varpi) \wedge \tau' \in tval\left(K,\sigma|\varpi'\right) \wedge\right.\\
&\qquad \left.\tau'' \in tval\left(K,\sigma|\varpi''\right)\right\} \cup tval\left(K,\sigma|\varpi';\varpi''\right) \cup \overline{\bigcap tval(K,\sigma|\Delta)}\\
\overset{(\star)}{=}\ &\bigcup tval(K,\sigma|\Gamma) \cup \left\{\tau'\tau_1\cdots\tau_n\tau'' \mid \tau_i \in tval(K,\sigma|\varpi) \wedge \tau' \in tval\left(K,\sigma|\varpi'\right) \wedge\right.\\
&\qquad \left.\tau'' \in tval\left(K,\sigma|\varpi''\right)\right\} \cup \overline{\bigcap tval(K,\sigma|\Delta)}\\
\overset{\text{def.}}{=}\ &\bigcup tval(K,\sigma|\Gamma) \cup tval\left(K,\sigma|\varpi';\varpi^*;\varpi''\right) \cup \overline{\bigcap tval(K,\sigma|\Delta)}
\end{aligned}$$

pull*:

$$\bigcup tval(K,\sigma|\Gamma) \cup tval\left(K,\sigma|\varpi^*;\varpi'\right) \cup tval\left(K,\sigma|\varpi;\varpi^*;\varpi'\right) \cup \overline{\bigcap tval(K,\sigma|\Delta)}$$

$$\begin{aligned}
&\stackrel{\text{def.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi^*;\varpi'\big) \cup \{\tau \in tval(K,\sigma|\varpi) \mid \neg finite(\varpi)\} \cup \\
&\qquad \big\{\tau\tau' \mid \tau \in tval(K,\sigma|\varpi) \wedge finite(\tau) \wedge \tau' \in tval\big(K,\sigma|\varpi^*;\varpi'\big)\big\} \\
&\stackrel{(\star)}{=} \bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi^*;\varpi'\big) \cup \overline{\bigcap tval(K,\sigma|\Delta)}
\end{aligned}$$

elim$^*{}_2$: This rule, which is similar to elim;, is incomplete. Its soundness follows from the soundness of elim; together with the fact that $tval\big(K,\sigma|\varpi'\big) \subseteq tval(K,\sigma|\varpi)$ implies $tval\big(K,\sigma|\big(\varpi'\big)^*\big) \subseteq tval(K,\sigma|\varpi^*)$.

dupl*:

$$\begin{aligned}
&\bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi^*;\varpi'\big) \cup tval\big(K,\sigma|\varpi^*;\varpi^*;\varpi'\big) \cup \overline{\bigcap tval(K,\sigma|\Delta)} \\
\stackrel{\text{def.}}{=} &\bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi^*;\varpi'\big) \cup \{\tau_1 \cdots \tau_n \mid \tau_i \in tval(K,\sigma|\varpi) \wedge \neg finite(\varpi_n)\} \cup \\
&\quad \{\tau_1 \cdots \tau_n \tau_{n+1} \cdots \tau_m \mid \tau_i \in tval(K,\sigma|\varpi) \wedge \neg finite(\varpi_m)\} \cup \\
&\quad \big\{\tau_1 \cdots \tau_n \tau_{n+1} \cdots \tau_m \tau' \mid \tau_i \in tval(K,\sigma|\varpi) \wedge finite(\varpi_i) \wedge \tau' \in tval\big(K,\sigma|\varpi'\big)\big\} \cup \\
&\quad \overline{\bigcap tval(K,\sigma|\Delta)} \\
= &\bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi^*;\varpi'\big) \cup \{\tau_1 \cdots \tau_n \mid \tau_i \in tval(K,\sigma|\varpi) \wedge \neg finite(\varpi_n)\} \cup \\
&\quad \big\{\tau_1 \cdots \tau_n \tau' \mid \tau_i \in tval(K,\sigma|\varpi) \wedge finite(\varpi_i) \wedge \tau' \in tval\big(K,\sigma|\varpi'\big)\big\} \cup \\
&\quad \overline{\bigcap tval(K,\sigma|\Delta)} \\
\stackrel{(\star)}{=} &\bigcup tval(K,\sigma|\Gamma) \cup tval\big(K,\sigma|\varpi^*;\varpi'\big) \cup \overline{\bigcap tval(K,\sigma|\Delta)}
\end{aligned}$$

Soundness and completeness for elim$^\omega{}_1$, pull$^\omega$ and dupl$^\omega$ follows from the proofs of elim$^*{}_1$, pull* and dupl*, and $tval(K,\sigma|\varpi^\omega) \supseteq tval(K,\sigma|\varpi^*)$ (for all ϖ). For elim$^\omega{}_2$ (incomplete), we have to consider the additional case of an infinite repetition of traces of ϖ'; also then, since all traces of ϖ' are traces of ϖ, we can conclude soundness. The second premise is then not needed. Rule elim$^\omega{}_3$ is similar and follows again because $tval(K,\sigma|\varpi^\omega) \supseteq tval(K,\sigma|\varpi^*)$.

cut:

$$\begin{aligned}
&\Big(\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\varpi) \cup \overline{\bigcap tval(K,\sigma|\Delta)}\Big) \cap \\
&\qquad \Big(\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\neg\varpi) \cup \overline{\bigcap tval(K,\sigma|\Delta)}\Big)
\end{aligned}$$

$$\overset{\text{def.}}{=} \Big(\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|\varpi) \cup \overline{\bigcap tval(K,\sigma|\Delta)}\Big) \cap$$
$$\Big(\bigcup tval(K,\sigma|\Gamma) \cup \overline{tval(K,\sigma|\varpi)} \cup \overline{\bigcap tval(K,\sigma|\Delta)}\Big)$$
$$\overset{\text{dist.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup \overline{\bigcap tval(K,\sigma|\Delta)} \cup \Big(tval(K,\sigma|\varpi) \cap \overline{tval(K,\sigma|\varpi)}\Big)$$
$$\overset{\text{compl.}}{=} \bigcup tval(K,\sigma|\Gamma) \cup \overline{\bigcap tval(K,\sigma|\Delta)}$$

closeSubsume: We have to prove that

$$\bigcup tval(K,\sigma|\Gamma) \cup tval(K,\sigma|(C,\mathcal{U})) \cup \overline{tval(K,\sigma|(C',\mathcal{U}')) \cap \bigcap tval(K,\sigma|\Delta)} = Traces$$

based on the assumption that $(C,\mathcal{U}) \rhd \big(C',\mathcal{U}'\big)$. The latter is, by Def. 5.12, equivalent to

$$\bigcup_\beta \big\{val\big(K,\sigma,\beta|\big(fpv(C) := \overrightarrow{v}\big) \circ U\big)(\sigma) \mid K,\sigma,\beta \models \bigwedge \big(C[\overrightarrow{v}/fpv(C)]\big)\big\} \supseteq$$
$$\bigcup_\beta \Big\{val\Big(K,\sigma,\beta|\Big(fpv\big(C'\big) := \overrightarrow{v'}\Big) \circ U'\Big)(\sigma) \mid K,\sigma,\beta \models \bigwedge \Big(C'[\overrightarrow{v'}/fpv(C')]\Big)\Big\}$$

which by Def. 5.7 is the same as writing $tval(K,\sigma|(C,\mathcal{U})) \supseteq tval\big(K,\sigma|\big(C',\mathcal{U}'\big)\big)$, or equivalently $tval(K,\sigma|(C,\mathcal{U})) \cup \overline{tval(K,\sigma|(C',\mathcal{U}'))} = Traces$, thus the equation to show holds.

close and close$\top^\infty$ are trivial.

closeUnsat: It suffices to show that $tval(K,\sigma|(C,\mathcal{U})) = \emptyset$, because then also

$$tval(K,\sigma|(C,\mathcal{U});\varpi) = \emptyset$$

and the whole sequent is valid. Since for all K,σ it holds that

$$\bigcup_\beta \big\{val\big(K,\sigma,\beta|\big(fpv(\text{false}) := \overrightarrow{v}\big) \circ Skip\big)(\sigma) \mid K,\sigma,\beta \models \bigwedge \big(Skip[\overrightarrow{v}/fpv(\text{false})]\big)\big\} \supseteq$$
$$\bigcup_\beta \Big\{val\Big(K,\sigma,\beta|\Big(fpv(C) := \overrightarrow{v'}\Big) \circ U\Big)(\sigma) \mid K,\sigma,\beta \models \bigwedge \Big(C[\overrightarrow{v'}/fpv(C)]\Big)\Big\}$$

based on the assumption that $(\text{false}, Skip) \rhd (C,\mathcal{U})$. This is equivalent to

$$\emptyset \supseteq \bigcup_\beta \Big\{val\Big(K,\sigma,\beta|\Big(fpv(C) := \overrightarrow{v'}\Big) \circ U\Big)(\sigma) \mid K,\sigma,\beta \models \bigwedge \Big(C[\overrightarrow{v'}/fpv(C)]\Big)\Big\}$$

and thus to $tval(K,\sigma|(C,\mathcal{U})) = \emptyset$ due to Def. 5.7. □

E Refactoring Models

This section contains abstract program models for the refactorings discussed in Sect. 6.3. All abstract program models are specified for *total correctness* (i.e., examples with loops contain variant specifications) and are automatically provable with REFINITY/KeY. For space reasons, we omit declarations of dynamic frame specification variables and abstract predicates and functions—they should be clear from the context. Furthermore, we abstain from declaring that ghost variables are disjoint from abstract frames and footprints. These declarations can be added generically: All ghost variables are disjoint from all abstract frames and footprints occurring in the code. Apart from that, we applied some minor beautifications to the code shown in this section. This is due to the following problems in KeY's JML support which existed at the time of writing this thesis:

- JML comments are bound to statements or expressions, and JML **`assume`** declarations, of which **`ae_constraint`** is an alias, are internally realized as block contracts. Therefore, after **`ae_constraint`** declarations, a block like "`{;}`" has to be added. The block may not be empty since otherwise, the contract is bound to the next statement. We recommend using the unobtrusive "`;`" statement.
- In JML quantifiers, the non-Java type **`any`** is not allowed. All **`ae_constraint`** declarations containing such quantifiers have to be added to REFINITY models as "relational preconditions" instead, which are translated to plain JavaDL.
- Ghost variables cannot be assigned expressions containing uninterpreted JavaDL functions and predicates. Therefore, instead of simply writing

  ```
  //@ ghost boolean threw = throwsExcE(\value(footprintE));
  ```

  one has to use a workaround. We use an artificial "ghost setter" AS which has to assign the ghost variable and may not complete abruptly. Its normal completion postcondition ensures that the ghost variable is set as desired.

Most abstract program models for the refactoring techniques discussed in this thesis are contained as examples in the REFINITY distribution[1]. Simply choose an "Abstract Execution" example from the standard KeY examples dialog.

[1] https://www.key-project.org/REFINITY/

```
/*@ ae_constraint
  @   \disjoint(frameA, frameB) &&
  @   \disjoint(frameA, footprintB) &&
  @   \disjoint(frameB, footprintA) &&
  @   \mutex(
  @     returnsA(\value(footprintA)),
  @     returnsB(\value(footprintB)))
  @   && \mutex(
  @     returnsA(\value(footprintA)),
  @     throwsExcB(\value(footprintB)))
  @   && \mutex(
  @     throwsExcA(\value(footprintA)),
  @     throwsExcB(\value(footprintB)))
  @   && \mutex(
  @     throwsExcA(\value(footprintA)),
  @     returnsB(\value(footprintB)))
  @   && (throwsExcA(\value(footprintA))
  @     || returnsA(\value(footprintA))
  @       ==> \disjoint(frameB, relevant))
  @   && (throwsExcB(\value(footprintB))
  @     || returnsB(\value(footprintB))
  @       ==> \disjoint(frameA, relevant));
  @*/

//@ assignable frameA;
//@ accessible footprintA;
/*@ exceptional_behavior requires
  @   throwsExcA(\value(footprintA)); */
/*@ return_behavior requires
  @   returnsA(\value(footprintA)); */
\abstract_statement A;

//@ assignable frameB;
//@ accessible footprintB;
/*@ exceptional_behavior requires
  @   throwsExcB(\value(footprintB)); */
/*@ return_behavior requires
  @   returnsB(\value(footprintB)); */
\abstract_statement B;
```

```
/*@ ae_constraint
  @   \disjoint(frameA, frameB) &&
  @   \disjoint(frameA, footprintB) &&
  @   \disjoint(frameB, footprintA) &&
  @   \mutex(
  @     returnsA(\value(footprintA)),
  @     returnsB(\value(footprintB)))
  @   && \mutex(
  @     returnsA(\value(footprintA)),
  @     throwsExcB(\value(footprintB)))
  @   && \mutex(
  @     throwsExcA(\value(footprintA)),
  @     throwsExcB(\value(footprintB)))
  @   && \mutex(
  @     throwsExcA(\value(footprintA)),
  @     returnsB(\value(footprintB)))
  @   && (throwsExcA(\value(footprintA))
  @     || returnsA(\value(footprintA))
  @       ==> \disjoint(frameB, relevant))
  @   && (throwsExcB(\value(footprintB))
  @     || returnsB(\value(footprintB))
  @       ==> \disjoint(frameA, relevant));
  @*/

//@ assignable frameB;
//@ accessible footprintB;
/*@ exceptional_behavior requires
  @   throwsExcB(\value(footprintB)); */
/*@ return_behavior requires
  @   returnsB(\value(footprintB)); */
\abstract_statement B;

//@ assignable frameA;
//@ accessible footprintA;
/*@ exceptional_behavior requires
  @   throwsExcA(\value(footprintA)); */
/*@ return_behavior requires
  @   returnsA(\value(footprintA)); */
\abstract_statement A;
```

Relational Postcondition

\result_1==\result_2

Figure E.1: The Slide Statements Refactoring

```
/*@ ae_constraint
  @   \disjoint(frameE, footprintP) &&
  @   \disjoint(frameP, footprintE) &&
  @   \disjoint(frameP, frameE) &&
  @   \disjoint(frameE, relevant) &&
  @   \disjoint(frameP, relevant) &&
  @   \mutex(
  @     throwsExcE(\value(footprintE)),
  @     throwsExcP(\value(footprintP)),
  @     returnsP(\value(footprintP)));
  @*/

if (/*@ assignable frameE;
      @ accessible footprintE;
      @ exceptional_behavior requires
      @   throwsExcE(
      @     \value(footprintE));
      @*/
    \abstract_expression boolean e
) {
    /*@ assignable frameP;
      @ accessible footprintP;
      @ exceptional_behavior requires
      @  throwsExcP(\value(footprintP));
      @ return_behavior requires
      @  returnsP(\value(footprintP));
      @*/
    \abstract_statement P;
    \abstract_statement Q1;
} else {
    /*@ assignable frameP;
      @ ... */
    \abstract_statement P;
    \abstract_statement Q2;
}
```

```
/*@ ae_constraint
  @   \disjoint(frameE, footprintP) &&
  @   \disjoint(frameP, footprintE) &&
  @   \disjoint(frameP, frameE) &&
  @   \disjoint(frameE, relevant) &&
  @   \disjoint(frameP, relevant) &&
  @   \mutex(
  @     throwsExcE(\value(footprintE)),
  @     throwsExcP(\value(footprintP)),
  @     returnsP(\value(footprintP)));
  @*/

/*@ assignable frameP;
  @ accessible footprintP;
  @ exceptional_behavior requires
  @  throwsExcP(\value(footprintP));
  @ return_behavior requires
  @  returnsP(\value(footprintP));
  @*/
\abstract_statement P;

if (/*@ assignable frameE;
      @ accessible footprintE;
      @ exceptional_behavior requires
      @   throwsExcE(
      @     \value(footprintE));
      @*/
    \abstract_expression boolean e
) {
    \abstract_statement Q1;
} else {
    \abstract_statement Q2;
}
```

Relational Postcondition

```
\result_1==\result_2
```

Figure E.2: The Cons. Dupl. Cond. Fragments Refactoring (Extract Prefix Variant)

```
/*@ ae_constraint (\forall any fp;
  @   (throwsExcP(fp) || returnsP(fp)));
  @*/

if (
    /*@ assignable \nothing;
      @ accessible footprintE1;
      @*/
    \abstract_expression boolean e1
) {
    /*@ assignable frameP;
      @ accessible footprintP;
      @ exceptional_behavior requires
      @   throwsExcP(
      @     \value(footprintP));
      @ return_behavior requires
      @   returnsP(\value(footprintP));
      @*/
    \abstract_statement P;
}

if (
    /*@ assignable \nothing;
      @ accessible footprintE2;
      @ exceptional_behavior requires
      @   false;
      @*/
    \abstract_expression boolean e2
) {
    /*@ assignable frameP;
      @ ...
      @*/
    \abstract_statement P;
}
```

```
/*@ ae_constraint (\forall any fp;
  @   (throwsExcP(fp) || returnsP(fp)));
  @*/

if (
    (/*@ assignable \nothing;
       @ accessible footprintE1;
       @*/
     \abstract_expression boolean e1) |
    (/*@ assignable \nothing;
       @ accessible footprintE2;
       @ exceptional_behavior requires
       @   false;
       @*/
     \abstract_expression boolean e2)
) {
    /*@ assignable frameP;
      @ accessible footprintP;
      @ exceptional_behavior requires
      @   throwsExcP(
      @     \value(footprintP));
      @ return_behavior requires
      @   returnsP(\value(footprintP));
      @*/
    \abstract_statement P;
}
```

Relational Postcondition

\result_1==\result_2

Figure E.3: The Consolidate Duplicate Expression (Consecutive Conditionals)

```
/*@ ae_constraint (\forall any fp;
  @   (throwsExcP(fp) || returnsP(fp)));
  @*/

if (
    /*@ assignable frameE1;
      @ accessible footprintE1;
      @*/
    \abstract_expression boolean e1
) {
    if (
        /*@ assignable frameE2;
          @ accessible footprintE2;
          @*/
        \abstract_expression boolean e2
    ) {
        /*@ assignable frameP;
          @ accessible footprintP;
          @*/
        \abstract_statement P;
    }
}
```

```
/*@ ae_constraint (\forall any fp;
  @   (throwsExcP(fp) || returnsP(fp)));
  @*/

  if (
    (/*@ assignable frameE1;
       @ accessible footprintE1;
       @*/
    \abstract_expression boolean e1) &&
    (/*@ assignable frameE2;
       @ accessible footprintE2;
       @*/
    \abstract_expression boolean e2)
) {
    /*@ assignable frameP;
      @ accessible footprintP;
      @*/
    \abstract_statement P;
}
```

Relational Postcondition

```
\result_1==\result_2
```

Figure E.4: The Consolidate Duplicate Expression (Nested Conditionals)

```
//@ ae_constraint \disjoint(var, frameP);        var = extracted(var);

//@ ghost Object oldVar = var;

/*@ assignable \hasTo(var), frameP;
  @ accessible footprintP;
  @ return_behavior requires false;
  @ exceptional_behavior ensures var == oldVar;
  @*/
\abstract_statement P;
```

Method-Level Context

```
private Object extracted(Object var) {
    Object res = var;
    /*@ ae_constraint
      @   \disjoint(var, frameP) &&
      @   \disjoint(res, frameP) &&
      @   \disjoint(res, relevant) &&
      @   \disjoint(res, footprintP);
      @*/

    //@ ghost Object oldVar = var;
    /*@ assignable \hasTo(res), frameP;
      @ accessible footprintP;
      @ return_behavior requires false;
      @ exceptional_behavior ensures res == oldVar;
      @*/
    \abstract_statement P;

     return res;
}
```

Relational Postcondition

```
\result_1==\result_2
```

Figure E.5: The Extract Method Refactoring

```
/*@ ae_constraint (\forall any fpB;
  @   (    returnsB(fpB)
  @     || throwsExcB(fpB)));
  @*/

res = mBefore();
```

```
/*@ ae_constraint (\forall any fpB;
  @   (    returnsB(fpB)
  @     || throwsExcB(fpB)));
  @*/

/*@ assignable frameA;
  @ accessible footprintA;
  @ return_behavior requires false;
  @*/
\abstract_statement A;

res = mAfter();
```

Method-Level Context

```
private Object mBefore() {
    /*@ assignable frameA;
      @ accessible footprintA;
      @ return_behavior requires false;
      @*/
    \abstract_statement A;

    /*@ assignable frameB;
      @ accessible footprintB;
      @ exceptional_behavior requires
      @   throwsExcB(\value(footprintB));
      @ return_behavior requires
      @   returnsB(\value(footprintB));
      @*/
    \abstract_statement B;
}
```

```
private Object mAfter() {
    /*@ assignable frameB;
      @ accessible footprintB;
      @ exceptional_behavior requires
      @   throwsExcB(\value(footprintB));
      @ return_behavior requires
      @   returnsB(\value(footprintB));
      @*/
    \abstract_statement B;
}
```

Relational Postcondition

```
\result_1==\result_2
```

Figure E.6: The Move Statements to Callers Refactoring

```
/*@ ae_constraint
  @   (\disjoint(
  @      frameTry, footprintCatch) ||
  @      ( \disjoint(
  @          frameCatch, relevant) &&
  @        (\forall any fp;
  @           (!throwsExcCatch(fp)
  @             && !returnsCatch(fp)))))
  @   && \disjoint(frameTry, relevant);
  @*/

try {
    /*@ assignable frameTry;
      @ accessible footprintTry;
      @ exceptional_behavior
      @   requires throwsExcTryStmt(
      @     \value(footprintTry));
      @*/
    \abstract_statement TryStmt;
} catch (Throwable t) {
    /*@ assignable frameCatch;
      @ accessible footprintCatch;
      @ exceptional_behavior requires
      @   throwsExcCatch(
      @     \value(footprintCatch));
      @ return_behavior requires
      @   returnsCatch(
      @     \value(footprintCatch));
      @*/
    \abstract_statement CatchProg;
}
```

```
/*@ ae_constraint
  @   (\disjoint(
  @      frameTry, footprintCatch) ||
  @      ( \disjoint(
  @          frameCatch, relevant) &&
  @        (\forall any fp;
  @           (!throwsExcCatch(fp)
  @             && !returnsCatch(fp)))))
  @   && \disjoint(frameTry, relevant);
  @*/

if (
    /*@ assignable \nothing;
      @ accessible footprintTry;
      @ normal_behavior
      @   ensures \result <==>
      @     !throwsExcTryStmt(
      @       \value(footprintTry));
      @ exceptional_behavior
      @   requires false;
      @*/
    \abstract_expression boolean e
) {
    /*@ assignable frameTry;
      @ ...
      @*/
    \abstract_statement TryStmt;
} else {
    /*@ assignable frameCatch;
      @ ...
      @*/
    \abstract_statement CatchProg;
}
```

Relational Postcondition

\result_1==**\result_2**

Figure E.7: The Replace Exception with Test Refactoring (Variant 1/2)

```
/*@ ae_constraint \disjoint(
  @   frameTry, footprintCatch);
  @*/

try {
    /*@ assignable frameTry;
      @ accessible footprintTry;
      @ exceptional_behavior
      @   requires
      @     throwsExcTryStmt(
      @       \value(footprintTry));
      @*/
    \abstract_statement TryStmt;
} catch (Throwable t) {
    /*@ assignable frameCatch,
      @   \hasTo(frameTry);
      @ accessible footprintCatch;
      @*/
    \abstract_statement CatchProg;
}
```

```
/*@ ae_constraint \disjoint(
  @   frameTry, footprintCatch);
  @*/

if (
    /*@ assignable \nothing;
      @ accessible footprintTry;
      @ normal_behavior
      @   ensures \result <==>
      @     !throwsExcTryStmt(
      @       \value(footprintTry));
      @ exceptional_behavior
      @   requires false;
      @*/
) {
    /*@ assignable frameTry;
      @ accessible footprintTry;
      @ exceptional_behavior
      @   requires throwsExcTryStmt(
      @     \value(footprintTry));
      @*/
    \abstract_statement TryStmt;
} else {
    /*@ assignable frameCatch,
      @   \hasTo(frameTry);
      @ accessible footprintCatch;
      @*/
    \abstract_statement CatchProg;
}
```

Relational Postcondition

\result_1==\result_2

Figure E.8: The Replace Exception with Test Refactoring (Variant 3)

```
/*@ ae_constraint \disjoint(
  @   footprintRollback, frameTry);
  @*/

try {
    /*@ assignable frameTry;
      @ accessible footprintTry;
      @ exceptional_behavior
      @   requires throwsExcTryStmt(
      @     \value(footprintTry));
      @*/
    \abstract_statement TryStmt;
} catch (Throwable t) {
    /*@ assignable \hasTo(frameTry);
      @ accessible footprintRollback;
      @*/
    \abstract_statement Rollback;

    /*@ assignable frameCatch;
      @ accessible footprintCatch;
      @*/
    \abstract_statement CatchProg;
}
```

```
/*@ ae_constraint \disjoint(
  @   footprintRollback, frameTry);
  @*/

if (
    /*@ assignable \nothing;
      @ accessible footprintTry;
      @ normal_behavior
      @   ensures \result <==>
      @     !throwsExcTryStmt(
      @       \value(footprintTry));
      @ exceptional_behavior
      @   requires false;
      @*/
    \abstract_expression boolean e
) {
    /*@ assignable frameTry;
      @ accessible footprintTry;
      @ exceptional_behavior
      @   requires throwsExcTryStmt(
      @     \value(footprintTry));
      @*/
    \abstract_statement TryStmt;
} else {
    /*@ assignable \hasTo(frameTry);
      @ accessible footprintRollback;
      @*/
    \abstract_statement Rollback;

    /*@ assignable frameCatch;
      @ accessible footprintCatch;
      @*/
    \abstract_statement CatchProg;
}
```

Relational Postcondition

```
\result_1==\result_2
```

Figure E.9: The Replace Exception with Test Refactoring (Rollback)

```
/*@ ae_constraint
  @   \disjoint(frameP, footprintG) &&
  @   \disjoint(frameQ, footprintG) &&
  @   \disjoint(frameP, frameQ) &&
  @   \disjoint(frameP, footprintQ) &&
  @   \disjoint(frameQ, footprintP);
  @*/

/*@ assignable \hasTo(footprintG);
  @ accessible \nothing;
  @ exceptional_behavior requires false;
  @ return_behavior requires false;
  @*/
\abstract_statement InitLoopCnt;

/*@ ae_constraint
  @   variant(\value(footprintG)) >= 0 &&
  @   loopInvG(\value(footprintG)) &&
  @   loopInvP(\value(frameP), \value(
  @     footprintP), \value(footprintG)) &&
  @   loopInvQ(\value(frameQ), \value(
  @     footprintQ), \value(footprintG));
  @*/

/*@ loop_invariant
  @   variant(\value(footprintG)) >= 0 &&
  @   loopInvP(\value(frameP), \value(
  @     footprintP), \value(footprintG)) &&
  @   loopInvQ(\value(frameQ), \value(
  @     footprintQ), \value(footprintG)) &&
  @   loopInvG(\value(footprintG));
  @ assignable frameP, frameQ, footprintG;
  @ decreases variant(\value(footprintG));
  @*/
while (
    /*@ assignable \nothing;
      @ accessible footprintG;
      @ normal_behavior ensures  \result <==>
      @   guardIsTrue(\value(footprintG));
      @ exceptional_behavior requires false;
      @*/
    \abstract_expression boolean e
) {
    /*@ ghost int oldVariant =
      @   variant(\value(footprintG)) */;

    /*@ assignable \hasTo(footprintG);
      @ accessible \nothing;
      @ normal_behavior ensures
      @   loopInvG(\value(footprintG)) &&
      @   variant(\value(footprintG)) >= 0 &&
      @   variant(\value(footprintG)) <
      @     oldVariant;
      @ exceptional_behavior requires false;
      @ return_behavior requires false;
      @ break_behavior requires false;
      @ continue_behavior requires false;
      @*/
    \abstract_statement LoopUpdate;

    /*@ assignable frameP;
      @ accessible footprintP, footprintG;
      @ normal_behavior ensures
      @   loopInvP(\value(frameP), \value(
      @     footprintP), \value(footprintG));
      @ exceptional_behavior requires false;
      @ return_behavior requires false;
      @ break_behavior requires false;
      @ continue_behavior requires false;
      @*/
    \abstract_statement P;

    /*@ assignable frameQ;
      @ accessible footprintQ, footprintG;
      @ normal_behavior ensures
      @   loopInvQ(\value(frameQ), \value(
      @     footprintQ), \value(footprintG));
      @ continue_behavior ensures
      @   loopInvQ(\value(frameQ), \value(
      @     footprintQ), \value(footprintG));
      @ exceptional_behavior ensures ...;
      @ return_behavior ensures ... &&
      @   returnedQ(\value(footprintQ));
      @ break_behavior ensures ... &&
      @   didBreakQ(\value(footprintQ));
      @*/
    \abstract_statement Q;
}
```

Figure E.10: The Split Loop Refactoring (Original Program)

```
/*@ ae_constraint
  @   \disjoint(frameP, footprintG) &&
  @   \disjoint(frameQ, footprintG) &&
  @   \disjoint(frameP, frameQ) &&
  @   \disjoint(frameP, footprintQ) &&
  @   \disjoint(frameQ, footprintP);
  @*/

/*@ assignable \hasTo(footprintG);
  @ ... */
\abstract_statement InitLoopCnt;

/*@ ae_constraint
  @   variant(\value(footprintG)) >= 0 &&
  @   loopInvG(\value(footprintG)) &&
  @   loopInvP(\value(frameP), \value(
  @     footprintP), \value(footprintG));
  @*/

/*@ loop_invariant
  @   variant(\value(footprintG)) >= 0 &&
  @   loopInvP(\value(frameP), \value(
  @     footprintP), \value(footprintG)) &&
  @   loopInvG(\value(footprintG));
  @ assignable frameP, frameQ, footprintG;
  @ decreases variant(\value(footprintG));
  @*/
while (
    /*@ assignable \nothing;
      @ accessible footprintG;
      @ normal_behavior ensures  \result <==>
      @   guardIsTrue(\value(footprintG));
      @ exceptional_behavior requires false;
      @*/
    \abstract_expression boolean e
) {
    /*@ ghost int oldVariant =
      @   variant(\value(footprintG)) */;

    /*@ assignable \hasTo(footprintG);
      @ ... */
    \abstract_statement LoopUpdate;

    /*@ assignable frameP;
      @ ... */
    \abstract_statement P;
}



/*@ assignable \hasTo(footprintG);
  @ ...;
  @*/
\abstract_statement InitLoopCnt;

/*@ ae_constraint
  @   variant(\value(footprintG)) >= 0 &&
  @   loopInvG(\value(footprintG)) &&
  @   loopInvQ(\value(frameQ), \value(
  @     footprintQ), \value(footprintG));
  @*/

/*@ loop_invariant
  @   variant(\value(footprintG)) >= 0 &&
  @   loopInvQ(\value(frameQ), \value(
  @     footprintQ), \value(footprintG)) &&
  @   loopInvG(\value(footprintG));
  @ assignable frameP, frameQ, footprintG;
  @ decreases variant(\value(footprintG));
  @*/
while (
    /*@ assignable \nothing;
      @ accessible footprintG;
      @ normal_behavior ensures  \result <==>
      @   guardIsTrue(\value(footprintG));
      @ exceptional_behavior requires false;
      @*/
    \abstract_expression boolean e
) {
    /*@ ghost int oldVariant =
      @   variant(\value(footprintG)) */;

    /*@ assignable \hasTo(footprintG);
      @ ... */
    \abstract_statement LoopUpdate;

    /*@ assignable frameQ;
      @ ... */
    \abstract_statement Q;
}
```

Figure E.11: The Split Loop Refactoring (Transformed Program)

Relational Precondition

```
(\exists any _frP, _fpP, _fpG;
  (\forall any frP, fpP, fpG; (
         (variant(fpG) >= 0 &&
          loopInvP(frP, fpP, fpG) &&
          loopInvG(fpG) &&
          !guardIsTrue(fpG))
    <==> (_frP == frP && _fpP == fpP && _fpG == fpG)
))) &&
(\exists any _frQ, _fpQ, _fpG;
  (\forall any frQ, fpQ, fpG; (
         (variant(fpG) >= 0 &&
          loopInvQ(frQ, fpQ, fpG) &&
          loopInvG(fpG) &&
          (!guardIsTrue(fpG) ||
             returnedQ(fpQ) || threwExcQ(fpQ) || didBreakQ(fpQ)))
    <==> (_frQ == frQ && _fpQ == fpQ && _fpG == fpG)
)))
```

Relational Postcondition

```
\result_1==\result_2
```

Figure E.12: The Split Loop Refactoring (Relational Pre- and Postconditions)

```
done = false;
//@ ghost int i = 0;
//@ ae_constraint \disjoint(loopLocs, done) && \disjoint(loopLocs, i);

/*@ ae_constraint
  @   decrExpr(\value(loopLocs)) >= 0 && loopInv(\value(loopLocs)) &&
  @   !doneCondition(\value(loopLocs), 0); */
/*@ loop_invariant decrExpr(\value(loopLocs)) >= 0 &&
  @   (done <==> doneCondition(\value(loopLocs), i)) &&
  @   loopInv(\value(loopLocs));
  @ assignable loopLocs; decreases decrExpr(\value(loopLocs)); */
while (!done &&
   (/*@ assignable \nothing; accessible loopLocs;
      @ normal_behavior ensures \result <==> guardVal(\value(loopLocs));
      @ exceptional_behavior requires false; */
    \abstract_expression boolean g)) {
    //@ ghost int oldDecrExpr = decrExpr(\value(loopLocs));
    //@ ae_constraint \disjoint(loopLocs, oldDecrExpr);
    if (
        /*@ assignable \nothing; accessible loopLocs;
          @ normal_behavior ensures (boolean) \result <==>
          @   doneCondition(\value(loopLocs), i+1);
          @ exceptional_behavior requires false; */
        \abstract_expression boolean cond
    ) {
        /*@ assignable loopLocs, done; accessible loopLocs;
          @ normal_behavior ensures done == true &&
          @   doneCondition(\value(loopLocs), i+1);
          @ exceptional_behavior requires false; // ... */
        \abstract_statement P;
    }

    /*@ assignable loopLocs; accessible loopLocs;
      @ normal_behavior ensures
      @   (done <==> doneCondition(\value(loopLocs), i+1)) &&
      @   loopInv(\value(loopLocs)) &&
      @   decrExpr(\value(loopLocs)) >= 0 &&
      @   oldDecrExpr > decrExpr(\value(loopLocs));
      @ exceptional_behavior requires false; //... */
    \abstract_statement Q;
    //@ set i = i++;
}
```

Figure E.13: The Remove Control Flag Refactoring (Original Program)

```
//@ ghost int i = 0; ae_constraint \disjoint(loopLocs, i);
/*@ ae_constraint decrExpr(\value(loopLocs)) >= 0 && loopInv(\value(loopLocs)) &&
  @   !doneCondition(\value(loopLocs), 0); */
/*@ loop_invariant decrExpr(\value(loopLocs)) >= 0 &&
  @   !doneCondition(\value(loopLocs), i) && loopInv(\value(loopLocs));
  @ assignable loopLocs; decreases decrExpr(\value(loopLocs)); */
while (!done &&
   (/*@ assignable \nothing; accessible loopLocs;
      @ normal_behavior ensures \result <==> guardVal(\value(loopLocs));
      @ exceptional_behavior requires false; */
    \abstract_expression boolean g)) {
    //@ ghost int oldDecrExpr = decrExpr(\value(loopLocs));
    //@ ae_constraint \disjoint(loopLocs, oldDecrExpr);
    if (
        /*@ assignable \nothing; accessible loopLocs;
          @ normal_behavior ensures (boolean) \result <==>
          @   doneCondition(\value(loopLocs), i+1);
          @ exceptional_behavior requires false; */
        \abstract_expression boolean cond
    ) {
        /*@ assignable loopLocs; accessible loopLocs;
          @ normal_behavior ensures doneCondition(\value(loopLocs), i+1);
          @ exceptional_behavior requires false; /*...*/ @*/
        \abstract_statement R;
        //@ ghost int oldI = i; ae_constraint \disjoint(loopLocs, oldI);
        /*@ assignable loopLocs; accessible loopLocs;
          @ break_behavior ensures
          @   i == oldI + 1 &&
          @   !doneCondition(\value(loopLocs), i+1) &&
          @   loopInv(\value(loopLocs)) &&
          @   decrExpr(\value(loopLocs)) >= 0 &&
          @   oldDecrExpr > decrExpr(\value(loopLocs));
          @ exceptional_behavior requires false; /*...*/ @*/
        \abstract_statement S;
    }
    /*@ assignable loopLocs; accessible loopLocs;
      @ normal_behavior ensures
      @   !doneCondition(\value(loopLocs), i+1) &&
      @   loopInv(\value(loopLocs)) && /*...*/ @*/
    \abstract_statement Q;
    //@ set i = i++;
}
```

Figure E.14: The Remove Control Flag Refactoring (Transformed Program)

Relational Precondition

```
(\exists any _frL; (\exists int _I; (
  (\forall any frL; (\forall int I; (
          (decrExpr(frL) >= 0 && loopInv(frL) &&
           !(!doneCondition(frL, I) && guardVal(frL)))
     <==> (_frL == frL && _I == I)
))))))
```

Relational Postcondition

```
\result_1==\result_2
```

Figure E.15: The Remove Control Flag Refactoring (Rel. Pre- and Postcondition)